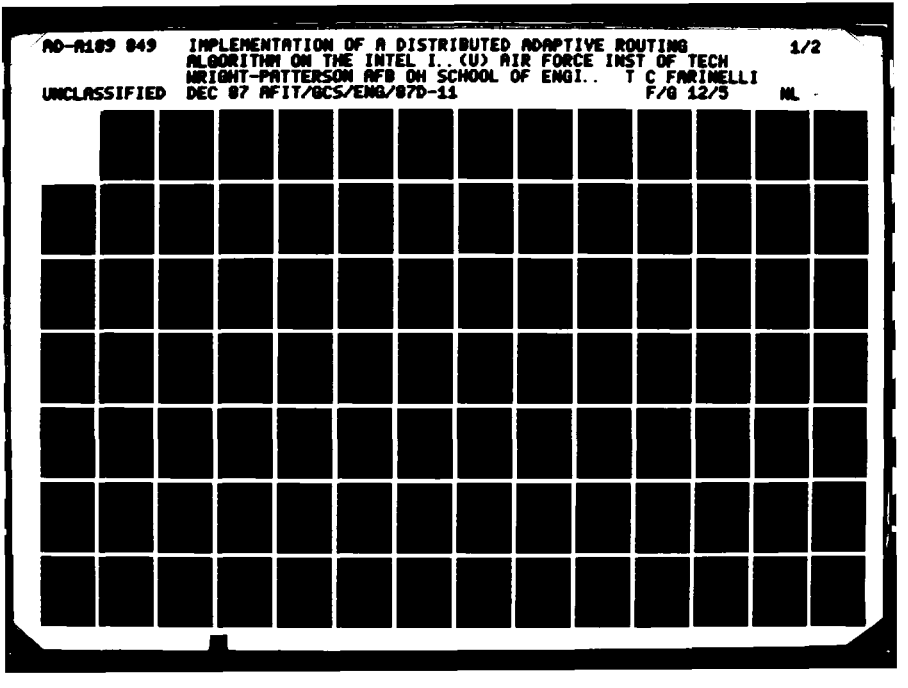
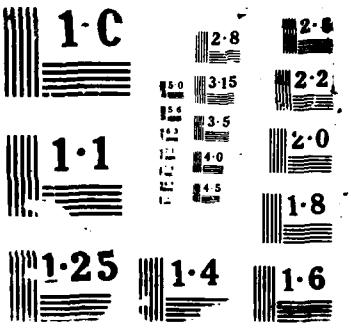

AD-A189 849 IMPLEMENTATION OF A DISTRIBUTED ADAPTIVE ROUTING ALGORITHM ON THE INTEL I..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T C FARINELLI
1/2
UNCLASSIFIED DEC 87 AFIT/GCS/ENG/87D-11 F/G 12/5 NL


1·C
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·8
3·15
3·5
4·0
4·5

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/87D-11

IMPLEMENTATION OF A DISTRIBUTED ADAPTIVE ROUTING ALGORITHM ON THE INTEL iPSC

THESIS

Tommy C. Farinelli
Captain, USAF

AFIT/GCS/ENG/87D-11

Approved for public release; distribution unlimited

AFIT/GCS/ENG/87D-11

IMPLEMENTATION OF A DISTRIBUTED ADAPTIVE ROUTING ALGORITHM ON THE INTEL iPSC

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Systems)

Tommy C. Farinelli, B.C.E.
Captain, USAF

December, 1987

Preface

The purpose of this study was to examine the performance of distributed adaptive routing algorithms on concurrent class computers. The Intel Personal Super-Computer (iPSC) was used as the test computer system. This study was limited to implementing the routing algorithm at the applications layer of the iPSC. Functions to interface between a user process and the routing process were written to imitate the current system message passing functions.

I am indebted to my faculty advisor, LtCol Charles Bisbee, for his guidance in the accomplishing of this research. I also wish to acknowledge my committee members, Capt Nathaniel Davis and Capt Bruce George, for their guidance and encouragement. Finally, I especially wish to thank my wife Cathy and my sons Jason, John, and Justin, for their understanding and support.

Tommy C. Farinelli

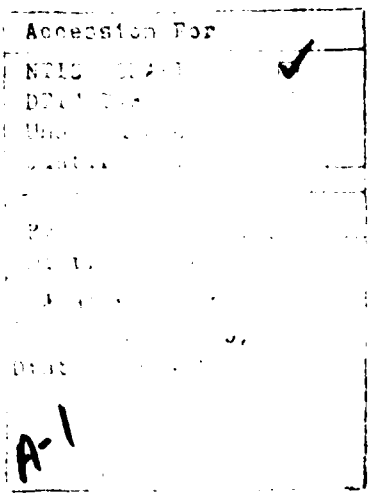


Table of Contents

List of Figures

List of Tables

Abstract

The purpose of this study was to examine the use of distributed adaptive routing algorithms on concurrent class computers. The Intel Personal SuperComputer (iPSC) was used as the test computer system. The implemented routing algorithm allowed each node to select the next node based on two criteria. The first criteria was the fewest number of hops; the second was the smallest delay time.

This study was limited to the comparison of a distributed adaptive routing algorithm, implemented at the applications layer, with the current static routing and with a simulation of the current routing implemented at the applications layer. The comparsion with the current routing algorithm provides a measure of the penalty for the implementation at the applications layer. The comparsion with the simulated current static routing provides a measure of the possible performance gain had the adaptive routing algorithm been implemented at the network layer.

In all three configurations were tested to formulate the comparisons. Each configuration was comprised of four processes: a Host Process, a Routing Process, a Ring Control Process, and a Network Loading Process. The Host Process controlled the loading of the processes onto the iPSC, the Routing Process controlled the message routing, the Ring Control Process provided the baseline message passing, while the Network Loading Process provided communications congestion on selected links. The metric used to compare the Routing Process performance was the average delay time for passing a message around the ring.

IMPLEMENTATION OF A DISTRIBUTED ADAPTIVE ROUTING ALGORITHM ON THE INTEL iPSC

I. Introduction

Purpose

The purpose of this thesis was to combine the study of distributed adaptive routing algorithms and concurrent processing. In particular, this study simulates a distributed adaptive routing algorithm on the Intel Personal SuperComputer (iPSC). Additionally, the developed program was modified to simulate the non-adaptive routing implemented on the iPSC. Therefore, a comparison of the two routing algorithms could be accomplished. A 32-node version of the iPSC was used as the vehicle for this research.

General Background

Connecting physically separated computer resources is an effective way to solve problems that require uneconomical amounts of time and/or resources on a single computer. John Stankovic, in his paper on distributed computer systems, states that "significant advantages, including good performance, good reliability, good resource sharing, and extensibility [16, 1102]" can be acheived through the use of multiple processors and an efficient communication network. An important factor of the communication network is the routing algorithm that is used to "determine the path a message follows from its source to its destination" [16, 1107].

Routing Algorithms. These algorithms are generally classified in two major categories: non-adaptive and adaptive [17, 198]. Non-adaptive (i.e., static) routing techniques are simple and easily developed, but they normally are not efficient

when communication loads vary on the communication paths [17, 199]. Message traffic congestion and malfunctions that occur during the operation of the system require alterations to the normal message routing. Adaptive (i.e., dynamic) routing techniques have the ability to adapt to a changing network environment making it possible to alter message routing based on current network communication loads. Adaptive routing techniques are further classified into centralized and distributed routing algorithms [17, 198]. Centralized routing is controlled by a central administrator that determines the best communication paths for each pair of source and destination nodes in the network.

Distributed Adaptive Routing. A distributed adaptive routing algorithm requires that each processor or node in a network have the ability to determine the route to a particular destination. To determine the routing, the algorithm, which is normally based on some performance metric, performs calculations on data that is commonly stored in a table or database format. The data in the routing table may represent a variety of information depending on the algorithm. According to Stankovic, "the metric might be number of hops, some estimate of delay to the destination, or buffer lengths" [16, 1107]. Also according to Stankovic:

> Such algorithms have the potential for good performance and reliability because the distributed control can operate in the presence of failures and quickly adapt to changing traffic patterns [16, 1107].

Supercomputing. The need for lower-cost supercomputing was expressed in an article by Justin Rattner, the Director of Technology for Intel Scientific Computers. Rattner explains, the supercomputer is "an essential tool for research, design, and development [12, 159]," but the cost of available supercomputers is too high for many universities and commercial users.

A second problem involving current supercomputers is the requirement for vector operation types and array data types. Operation types are generally divided

between scalar and vector operations. Rattner states, "existing supercomputers are essentially vector processors," and these machines depend on their data "being in the form of either vectors or arrays" [12, 159]. Vector processors efficiently handle vector operations, but "a portion of any code will consist of scalar (single-quantity) operations [12, 159]," these scalar operations create performance bottlenecks. Therefore, these scalar operations force vector processors to operate at a fraction of their peak performance [12, 161]. Because of operations and data type constraints, "either programmers or sophisticated compilers [12, 160]," are required to vectorize the code. These requirements are part of additional overhead necessary to achieve optimal performance [12, 160].

Concurrent Processing. Concurrent processing offers solutions to these problems. Rattner defines concurrency as "a high-level or global form of parallelism, denoting independent operation of a collection of simultaneous computing activities" [12, 160]. Therefore, a complex problem, that can be separated into a number of smaller simpler problems, can be solved simultaneously on a concurrent machine that "uses loosely coupled, multiple, interacting processors" [12, 160].

The sharing of the load by the multiple processors in a concurrent architecture aids in achieving high computational efficiency. Significant cost/performance benefits are achieved over vector processors that cannot operate at their peak performance [12, 161].

The cost benefit of a concurrent machine is achieved through the cost reduction provided by very large scale integration (VLSI) advances and through the use of off-the- shelf components verses custom built special purpose components. The reduced cost allows for a larger number of units to be sold which tends to further reduce the costs [12, 162].

The multiple processors can not perform their tasks in total isolation, there must be high speed communications available so data can be exchanged when nec-

essary. Intel's Personal SuperComputer (iPSC) uses a static routing technique that inhibits the passing of data when one communication path becomes congested with messages. The implementation of a distributed adaptive routing algorithm should provide a substantial increase in the throughput of the network under a communications bound condition.

Organization

Chapter II includes an overview of computer network routing algorithms, a taxonomy for routing algorithms, and a study of several distributed routing algorithms. The routing algorithms studied are a research network developed by K. Brayer of the Mitre Corporation, Digital Equipment Corporation's Digital Network Architecture (DNA), and the Advanced Research Projects Agency's Network (ARPANET).

Chapter III examines the current message passing structure of the iPSC. The examination includes discussions on the current routing algorithm, the current hardware, and situations in which message passing is impeded on the iPSC.

Chapter IV contains a discussion about the adaptive routing algorithm that was implemented, Appendix A and Appendix B contain the source code.

Chapter V contains the test plan, configurations, and procedures used to measure the effect of the implemented adaptive routing algorithm on the iPSC.

Chapter VI contains the results and analysis of the data obtained from the testing performed in Chapter V. The chapter concludes with recommendations for future study.

II. Distributed Adaptive Routing Background

This overview of distributed adaptive routing algorithms for computer networks begins with a look at the International Standards Organization's (ISO) model for connecting heterogenous computers in a network. The second section of this chapter consists of a taxonomy for categorizing computer networks and a general overview of routing algorithms used in computer networks. The third section describes the way three operational networks use distributed adaptive routing algorithms. The first network is a research network developed by K. Brayer at Rome Air Development Center. The second network is Digital Equipment Corporation's (DEC) Digital Network Architecture (DNA). The last algorithm discussed is used for the Advanced Research Projects Agency Network (ARPANET). The chapter concludes with a summary of distributed adaptive routing algorithms.

Open Systems Interconnection Model

The reference model for Open Systems Interconnection (OSI) supported by the ISO is a seven layer hierarchical view of computer networks [10, 144]. Figure 1 depicts the different layers that were developed to "decompose data communications into manageable pieces with well-defined interfaces" [10, 144]. While layer 1 supports the actual physical connection of two or more nodes in a network, the remaining layers are defined such that each layer has a "virtual connection to its distant peer" while only exchanging information with layers above and below the layer of interest [10, 144]. The network layer of the OSI model is the layer of interest for this research.

Tanenbaum states, the network layer, also known as layer 3, "controls the operation of the subnet" [17, 17]. In the OSI model, the network layer of the source node accepts a message from layer 4, divides the message into packets, and then routes the packets toward their destination [17, 18]. At an intermediate node, layer 3

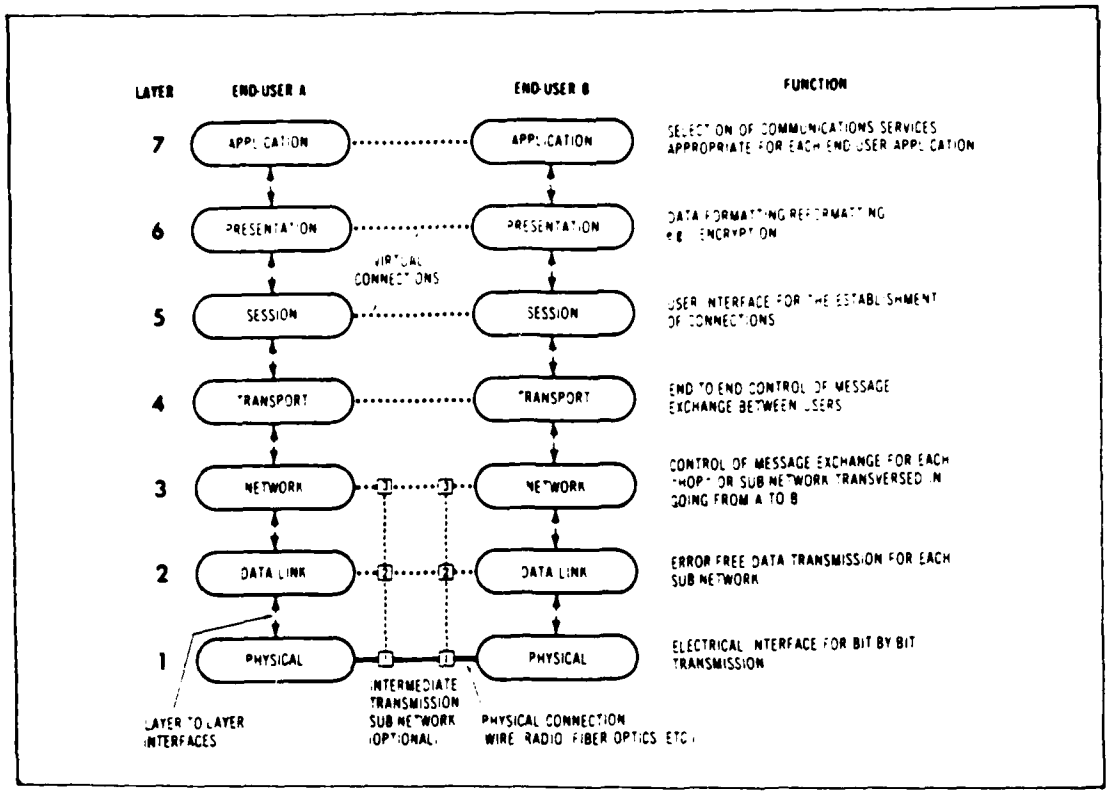


Figure 1. Reference Model for Open Systems Interconnection Source: [10, 145]

determines the next portion of the message's route. At the destination node, the network layer passes the packets to layer 4, also known as the transport layer [17, 18].

Distributed Routing Overview

The routing algorithm contained in layer 3 of the OSI model is an important factor in determining how the network will react to changes in topology and fluctuations in traffic load [17, 197]. A wide range of routing algorithms have been developed to support network performance objectives. The reminder of this section reviews several routing algorithm characteristics.

Routing Taxonomy. H. Rudin developed a taxonomy for adaptive routing algorithms that classifies routing algorithms by centralized vice distributed techniques

and complexity, see Figure 2. Rudin defines centralized techniques as those, "in which routing strategies are prepared centrally and then sent to the nodes for execution" [13, 44]. He defines distributed techniques as those in which "the strategy is prepared throughout the network" [13, 44]. Since this research is concerned with distributed adaptive routing algorithms the remaining focus is on Rudin's distributed techniques. According to Rudin, the two distributed techniques on the bottom of Figure 2 are simple to design, but they are also rather inefficient. The simplest distributed technique, known as *flooding*, transmits traffic on all outgoing lines. Another simple technique, known as *random routing*, transmits traffic on a randomly selected outgoing line, irregardless of the best path to the destination node. Moving up in complexity, Rudin's two isolated techniques are based on Baran's *hot potato* and *backward learning* algorithms. Bias is added to the shortest output queue, allowing an additional weight to be added to the calculations for the queues in Baran's *hot potato* algorithm and symmetrical traffic is assumed in the *backward learning* algorithm [13, 44].

Rudin's most complex class of algorithms is the cooperative class that includes periodic and asynchronous updates. In this technique the nodes report information about their own status to all other nodes in the network. This information normally includes the length of its output queues, time delay for message transmissions, or the number of nodes between itself and the other nodes in the network [13, 44].

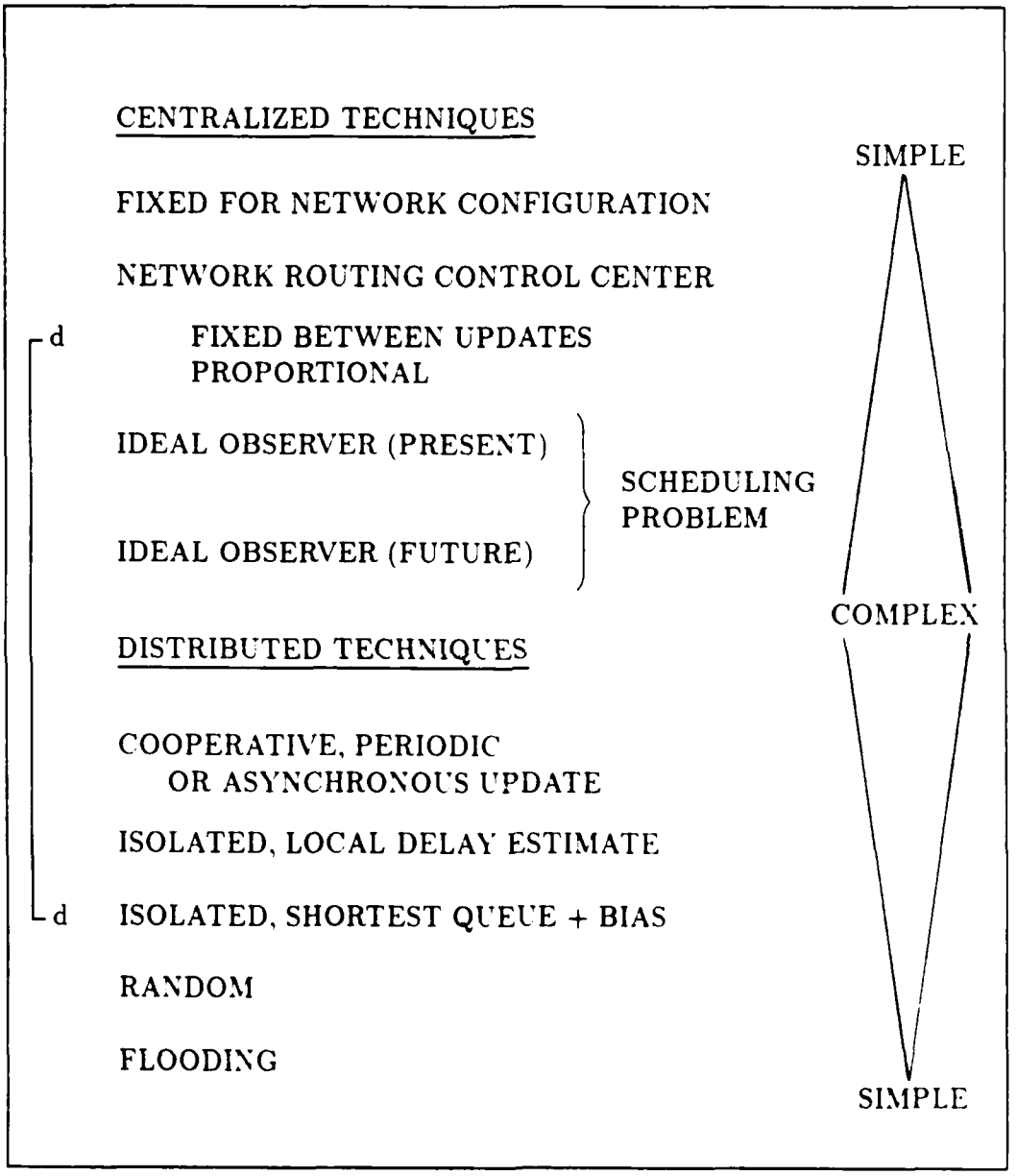


Figure 2. Routing Algorithm Classification Source: [13, 44]

Distributed Routing Algorithms. Tanenbaum, in his book *Computer Networks*, categorizes routing algorithms in a different manner than does Rudin. Tanenbaum separates routing algorithms into two groups: nonadaptive and adaptive. Nonadaptive routing algorithms cannot alter their connectivity in response to changes in network traffic, but adaptive routing algorithms can alter their connectivity. He further subdivides adaptive routing into centralized and non-centralized algorithms [17, 198].

Centralized routing normally utilizes a directory or table that contains information on how to forward messages between processors. The table is maintained at a routing control center (RCC) that receives information on the current status of the network from the other nodes in the network. Using this periodic information, the RCC makes the decisions determining the most efficient routes for traffic flow though the network. The routing information is then redistributed to the other nodes in the network [17, 201].

Unlike centralized algorithms, non-centralized algorithms maintain redundant information at each node. The node's ability to maintain their own information makes the network more robust and fault tolerant [17, 205]. Tanenbaum divides non-centralized algorithms into isolated and distributed routing algorithms [17, 201].

Isolated algorithms are differentiated from distributed algorithms by their information gathering techniques. Isolated algorithms base their routing decisions on information gained by analyzing traffic that passes through the node. Distributed algorithms use specific routing information exchanged between the nodes in the network [17, 202].

Tanenbaum discusses two types of isolated routing algorithms: Baran's *hot potato* and Baran's *backward learning* [17, 202]. The *hot potato* algorithm places the outgoing message in the shortest outgoing queue. The algorithm assumes that the message will arrive at its destination sooner by leaving the current node sooner on a randomly chosen route, than it would by waiting for a preselected route [17, 202].

The *backward learning* algorithm uses information from incoming messages to determine the best route for outgoing messages [17, 203]. One technique discussed by Tanenbaum requires each message to include identification of the first node capable of altering the message route and a counter that is incremented at every additional node capable of altering the route. These two pieces of data can be used by each node in the path to calculate the number of hops to the message source [17, 204].

The distributed routing algorithm discussed by Tanenbaum requires that a table be maintained at every node instead of a RCC. The table contains the address and preferred outgoing line for each node in the network, as well as, some measure of the time it takes to get to the destination [17, 205]. The distributed routing techniques enable more efficient use of resources than do the isolated routing techniques, but they require more processing overhead and they add additional message traffic to the network [13, 45].

Performance Objectives. By determining which nodes and links will be used to exchange information, the routing algorithm is an important factor in determining if the network can meet its performance objectives [5, 7]. Computer network performance objectives are based on speed and service of the network [9].

The service objective is based on four parameters: availability, data integrity, message integrity, and security [9]. Tanenbaum places the service objectives into the network reliability performance constraint [17, 34]. The speed objective is measured in terms of minimum delay and maximum throughput [9].

Unfortunately the two parameters, delay and throughput, are mutually exclusive. Delay is defined as the "time between transmission of the first bit and delivery of the last bit of a message" [9] and is measured in terms of "mean packet delay not exceed a given number of milliseconds" [17, 34]. While throughput is defined as the "number of bits sent divided by time between transmission of the first bit and delivery of the last bit" [9]. When the throughput of the network is increased the

delay of an individual message will increase.

Operational Networks

As stated earlier, this section reviews three operational networks that use a form of distributed adaptive routing for their routing algorithms. While not adding additional theoretical information, this section was included to give expanded background in the ways distributed adaptive routing algorithms have been implemented in operational networks.

Brayer's Research Network. K. Brayer of the Mitre Corporation developed a research packet switch system that is loop-free and survivable. His algorithm is divided into a mathematical algorithm to determine the shortest path and a set of procedures (message routing and address finding sections) to control or utilize the results of the mathematical algorithm [4, 93].

Initially, each node transmits on its outgoing lines a special start up identification message telling its nearest neighbors its identification (ID), while scanning incoming lines to learn its nearest neighbor's ID. After the initialization period, each node enters the message routing section of the algorithm and message transmissions may begin. If the location of a message addressee is unknown, the algorithm enters the address finding section to determine how to route the message [4, 94].

In the address finding section, the node holds the message while it transmits a *header message* to its nearest neighbors. This *header message* requests a response if the receiving node has the unknown address. A positive response is indicated by an acknowledgement that the message may be sent. If the address is not known at the neighbor node, the node adds its address onto the *header message* and transmits the message to its neighbors that have not seen the message (indicated by the lack of their address at the end of the message). When the addressee is located the *header message* returns to the original node (via the list of addresses at the end

of the *header message*). As the message retraces its course, all the nodes update their routing table with the new routing information. The *header message* can be retransmitted in the network a set number of times (normally determined by network size), when that number is reached without a positive response the original node is notified and queried if the sender wants to try later or let the network store the message and automatically send the *header message* at a later time [4, 94].

The message routing section forwards messages by first checking its routing table for an existing path. If a path exists the message is transmitted using the path. Otherwise, the message routing section is able to randomly select a node and forward the message to it. This random routing occurs in addition to the address finding section attempting to determine a path [4, 94]. If a message timeout occurs, the node can select a different node and retransmit the message or store the message and retransmit at a later time. When the number of timeouts exceeds a specified value, the routing tables are updated to avoid those links.

Figure 3 shows is an example of a network, stages of its connectivity matrix, and its distance matrix. The source node is labeled down the left edge with the destination labeled across the top. The distinction between source and destination is important only when the links are unidirectional [4, 95]. As the figure shows each node identifies its nearest neighbor with a path of length one. The remaining paths are filled in one row at a time. A maximum of $N - 2$ iterations are required to fill in the matrix, any remaining holes are caused by a disjointed network [4, 96]. The development of the connectivity and distance matrix forms the mathematical portion of the routing algorithm.

This protocol is considered survivable because each node can determine routes on its own, but the protocol can not promise to use the shortest path every time. Although, if the network is stable for a reasonable length of time, it will determine the shortest path [4, 95].

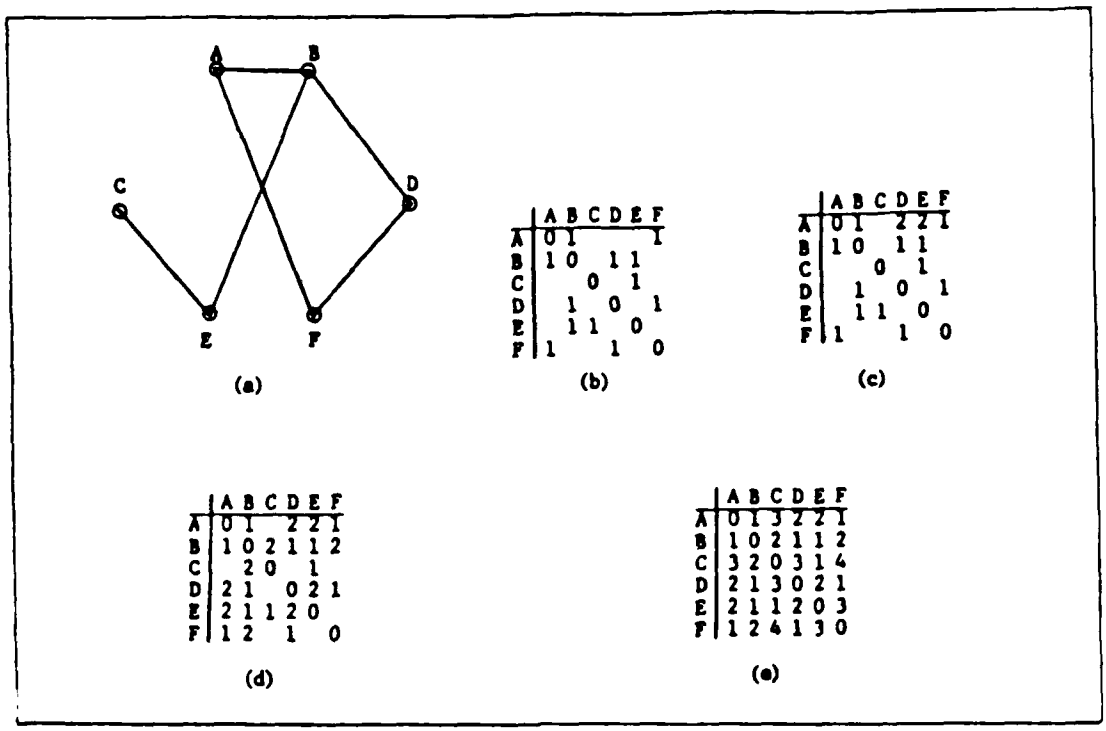


Figure 3. Example Network, Connectivity Matrix, and Distance Matrix Source: [4, 471]

Digital Network Architecture. Digital Equipment Corporation's (DEC) Digital Network Architecture (DNA) is implemented in a five layer hierarchy verses the seven layer ISO model. The layers of the hierarchy are called the physical link, data link control, transport, network services, and application [18, 516]. In Stuart Wecker's description of DEC's DNA, he explains the routing algorithms are defined as part of the transport layer. The transport layer also controls congestion and message lifetime for the network [18, 520].

DNA's routing algorithm is based on information stored in two $n \times m$ matrices, where n is the number of other nodes in the network and m is the number of output channels for the node. One matrix, called the *HOPS* matrix, contains the number of hops from the current node to the other nodes in the network via its output channels [18, 520]. The other matrix, called the *COST* matrix, is used to maintain the path

cost from the current node to the other nodes in the network [18, 520].

The cost of the path is inversely proportional to the quality of the path [18, 520]. Resource availability and processing capacity of the nodes; along with delay, throughput, and error rate characteristics of the lines are used to determine the *COST* matrix [18, 520].

The best paths in the network are determined by a comparison of the two matrices with each of the neighbors' matrices. For example, node A passes its best paths to its nearest neighbors, if a neighbor detects an improvement they update their matrices. After the neighbors update their matrices, they send the new information to their nearest neighbors, where the process is repeated.

The updating of the best paths can create looping in the network. When a loop is detected, a routing message is generated marking the node unreachable. A loop is detected when the longest number of hops in the network is exceeded by an entry in the *HOPS* matrix or the value of the *visit count* field of the routing header. The contents of the *visit count* field are incremented by one at each node the message reaches [18, 521].

ARPANET. The development of distributed routing algorithms was led by the Advanced Research Projects Agency when it developed the ARPANET. McQuillan, Richer, and Rosen describe problems with the original algorithm and changes used to correct some of these problems. They state:

> The new routing algorithm is an improvement over the old one in that it uses fewer network resources, operates on more realistic estimates of network conditions, reacts faster to important network changes, and does not suffer from long- term loops or oscillations [11, 712].

The algorithm is based on Dijkstra's form of the shortest-path-first (SPF) routing algorithm. In distributed fashion each node calculates the best paths to all other nodes in the network. The calculation uses a database that is maintained at

each node. The database contains information describing the network topology and the network line delays. The database is updated every 10 seconds via *routing update* messages that announce significant changes in the delays for the node's outgoing lines [11, 712].

The database contains two data structures, a tree and a list. The list structure contains nodes that are adjacent to the nodes currently on the tree. Initially, the tree structure consists of the current node as the root node. Since each node identifies its outgoing line delays, new nodes are added to the leaves of the tree structure by calculating the smallest total delay time. The algorithm builds a shortest path tree by cycling through the algorithm until all the nodes are accounted for [11, 712].

The new algorithm's delay measurement is calculated by time-stamping each packet when it arrives, when the first bit is sent, and when the acknowledgement is received. The first-bit-sent time is overwritten when the packet is retransmitted. After the acknowledgement is received, the first-bit-sent is subtracted from the arrival time. The delay for the packet is calculated by adding the above difference with the constant line propagation delay and the transmission delay, which is a factor of the packet length and line speed [11, 714].

An average of all the packet's delays is computed every 10 seconds. The average is then compared with the last average reported. If the difference exceeds a certain threshold the new delay is reported to the entire network [11, 714].

Problems still exist with the ARPANET routing procedure. Spinelli in his thesis states, it is still difficult to determine optimum settings for "the maximum age of packets, the minimum interval between creation of update packets, the maximum interval between creation of update packets, etc." [15, 19].

Summary

This chapter has given a broad overview of distributed routing algorithms. It has illustrated a taxonomy that can be used to classify and compare algorithms.

It also presented an overview of the some techniques used for distributed routing algorithms. The chapter concluded with a review of several operational routing algorithms.

In conclusion, some common threads run through all the distributed adaptive routing algorithms. They include a means to measure certain parameters of interest, a means to make a routing decision based on these measurements, and a means to inform the other nodes in the network of their decisions.

III. iPSC Background

This chapter overviews the characteristics of the Intel iPSC architecture, hardware, and routing algorithm. The first section provides an overview of concurrent architecture by presenting Flynn's taxonomy, two processor-memory configurations, and finally (possibly the most important for this thesis) an overview of interconnection networks. The second section of this chapter is a brief history and overview of the Intel iPSC. With the background presented the remainder of the chapter describes the current operation of the iPSC's message routing system.

Overview of Concurrent Architecture

The classic taxonomy for classifying computer systems was developed by Flynn in his paper of 1966. Siegel in his text *Interconnection Networks for Large-Scale Parallel Processing*, adds additional classification of concurrent machines by classifying the processor-memory configuration, as well as, interconnection networks. And finally, Feng in his survey of interconnection networks discusses various geometries for internodal communication (i.e., the interconnection network). These three areas are developed in more detail in the following section with emphasis on the categories that describe the iPSC.

Flynn's Taxonomy. Flynn formulated four computer organizations based on two definitions, the *Instruction Stream* and the *Data Stream*. He defines them as follows:

> Instruction Stream is the sequence of instructions as performed by the machine; Data Stream is the sequence of data called for by the instruction stream (including input and partial or temporary results) [7, 1902].

Using these two definitions Flynn labels four computer organizations as:

1) Single Instruction Stream-Single Data Stream (SISD)
2) Single Instruction Stream-Multiple Data Stream (SIMD)
3) Multiple Instruction Stream-Single Data Stream (MISD)
4) Multiple Instruction Stream-Multiple Data Stream(MIMD) [7, 1902].

Of these four labels the iPSC is best described as an MIMD computer [12, 163].

Processor-Memory Configuration. Siegel further defines an MIMD machine as, a system of "N processors, N memory modules, and an interconnection network [14, 30]." As shown in Figure 4 and Figure 5, processor-memory configurations

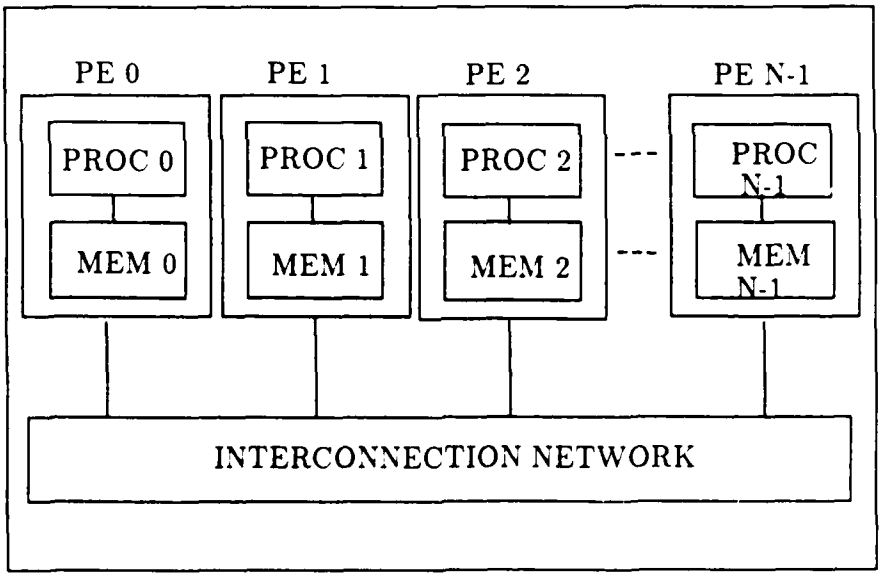


Figure 4. Processing-Element-to-Processing-Element MIMD-Machine Configuration with N Processing Elements Source: [14, 31]

typically come in two varieties. The first being a processing-element-to-processing-element (PE-to-PE), where the processing element is formed by a processor and

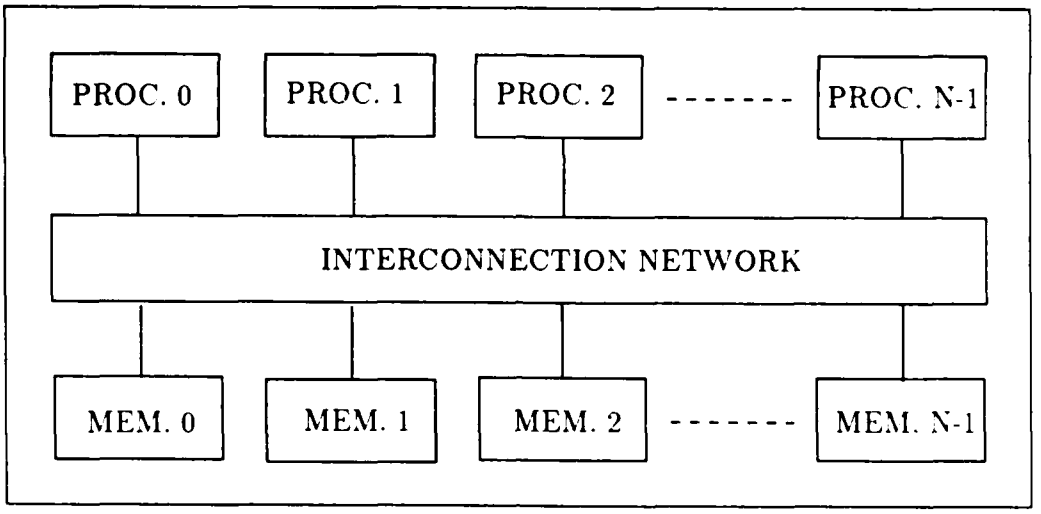


Figure 5. Processor-to-Memory MIMD-Machine Configuration with N Processors and N Memories Source: [14, 31]

memory pair and the interconnection network connects each independent element. And the second being a processor-to- memory configuration, where the interconnection network connects the N processors to the N memories [14, 30]. The iPSC uses a PE-to-PE configuration as shown, in Figure 6, by the block diagram of an iPSC's processing element. Each PE or node is centered around Intel's 80286 central processing unit and also consists of the 80287 numeric processor, 512K Bytes of dynamic RAM, and eight 82586 communication coprocessors [2, 6].

Interconnection Networks. Two basic categories of interconnection networks are single-stage and multi-stage. Multi-stage networks enable passing of data from its source directly to its destination, where as, in the single stage network data may have to recirculate through the stage several times to reach its destination [14, 20]. Siegel describes four configurations for single-stage networks used in both SIMD and

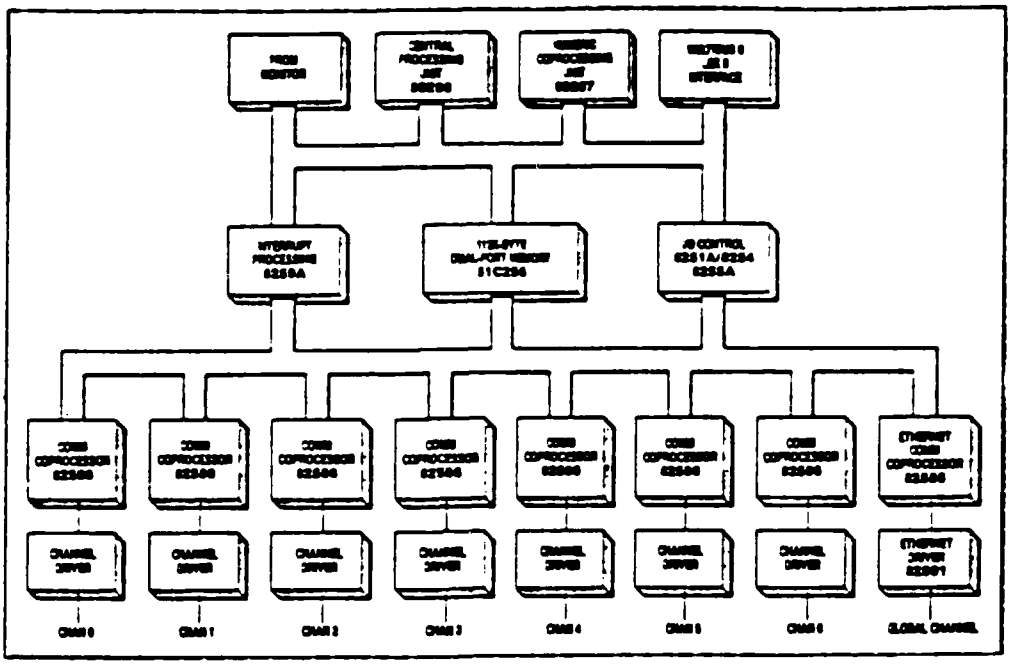


Figure 6. Block Diagram of an iPSC Processing Element Source: [2, 6]

MIMD machines. The configurations are the Illiac, the plus-minus 2^i (PM2I), the shuffle-exchange, and the cube. The cube, also known as an indirect binary n-cube or hypercube, is the configuration used in the iPSC [12, 163].

Each of the configurations can be defined mathematically by an interconnection function. The Illiac interconnection function is defined by four function as follows:

$$Illiac_{+1}(P) = P + 1 \bmod N \tag{1}$$

$$Illiac_{-1}(P) = P - 1 \bmod N \tag{2}$$

$$Illiac_{+n}(P) = P + n \bmod N \tag{3}$$

$$Illiac_{-n}(P) = P - n \bmod N \tag{4}$$

where

N = number of nodes in the network

n = square root of N

P = current processor [14, 22].

Therefore, the topology of the nodes appears as an n-by-n array [14, 22].

The PM2I interconnection function is defined by 2m functions, where m is the number of bits necessary to represent the number of nodes (N) in a binary number.

$$PM2_{+i}(P) = P + 2^i \bmod N \tag{5}$$

$$PM2_{-i}(P) = P - 2^i \bmod N \tag{6}$$

where

N = number of nodes in the network

m = $\log_2 N$

$0 \leq i < m$ [14, 23].

The shuffle-exchange interconnection function is defined by two functions. The shuffle equation is:

$$shuffle(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-2}p_{m-3}\cdots p_1p_0p_{m-1} \tag{7}$$

and the exchange equation is:

$$exchange(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-1}p_{m-2}\cdots p_1\overline{p}_0 \tag{8}$$

where

m = $\log_2 N$ [14, pages 24,25].

The cube network consists of m interconnection functions (m is also known as the dimension of the cube) and is defined by:

$$cube_i(p_{m-1}\cdots p_{i+1}p_ip_{i-1}\cdots p_0) = p_{m-1}\cdots p_{i+1}\overline{p}_ip_{i-1}\cdots p_0 \tag{9}$$

where

N = number of nodes

m = $\log_2 N$ [14, 26].

As shown in Figure 6, the iPSC system can attain a cube dimension of seven containing 128-nodes. An example of a 3 dimension cube is shown in Figure 7.

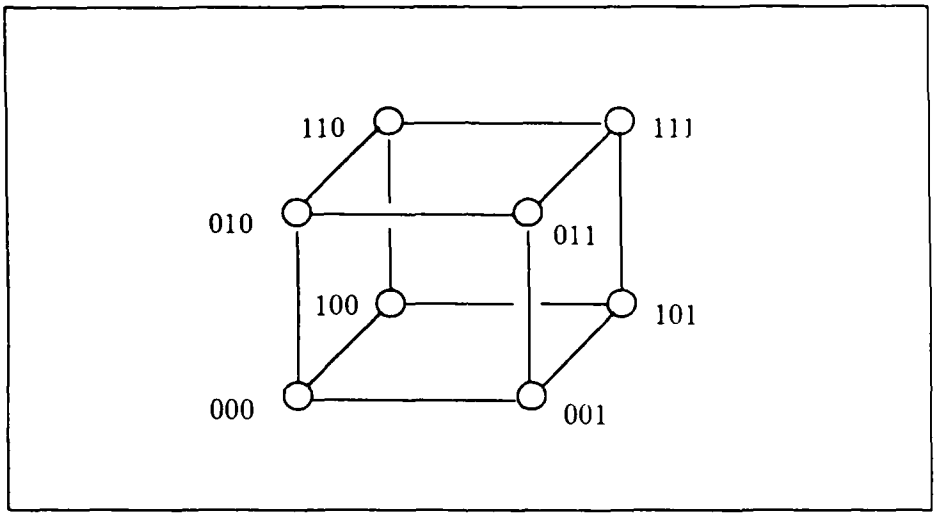


Figure 7. Three-Dimension Cube Structure

The communication channels are represented by edges in the graphical picture, while the nodes form the switching points [6, 110]. The nodes or processors in the iPSC are assigned binary numbers that serve as their *node addresses*. Using a gray code numbering scheme, bidirectional links connect nodes whose addresses vary by one binary digit.

Feng discusses interconnection networks and their role in providing communication paths between processors and memory modules [6, 109]. Feng states the selection of the proper architecture for an interconnection network should be based

on the operation mode, the control strategy, the switching method, and the network topology [6, 109].

History and Overview of the iPSC

The Intel iPSC concurrent computer is based on work performed by personnel at the California Institute of Technology and NASA's Jet Propulsion Laboratory. The Mark I Hypercube is a 64-node PE-to-PE system that utilized Intel 8086/8087 processors and a multibus interboard (processor) connection structure. Refinement of the Mark I led to the Mark II concurrent machine. The Mark II system uses a modularized design permitting one to four groupings of 32 PEs for a total of 128 PEs. The multibus interboard connection of the Mark I was replaced with a single-stage cube interconnection network. The iPSC further refined the Mark II machine by upgrading to the 80286/287 processors and refining the mechanism for passing messages [8, 353].

Access to the PEs of the iPSC is controlled by an Intel 310 machine, referred to as the Cube Manager. The Cube Manager provides a medium for uploading and downloading processes(programs) and data to the individual nodes. Rattner explains that, the iPSC interprocessor communications is controlled by a Node Operating System. The operating system "provides the system calls that enable processes to send and receive messages" [12, 164]. In addition, message routing, node-to-Cube Manager I/O, and process debugging are handled by the operating system.

Current Message Routing. Currently, message routing at the node level of the iPSC is controlled in a non-adaptive manner by software interrupts of the 80286 central processing unit, and is supported at the physical level by a pair of special purpose integrated circuits. As shown in Figure 6, each node of the iPSC includes eight Intel 82586 Local Area Network (LAN) Coprocessors. Seven of the 82586s are dedicated to interprocessor communication, while the eighth 82586 supports a global

communication link. Therefore, each node has the capability for connection to seven other nodes on communication channels zero to six. The 82586 LAN Coprocessor is matched with Intel's 82501 Ethernet Serial Interface chip, so that the IEEE 802.3/Ethernet specification is realized [3, 1-2]. Layers one and two of the ISO's Reference Model for Open Systems Interconnection (OSI) are implemented, as shown in Figure 8, by the use of the 82586 and the 82501 chips [1, 3-5].

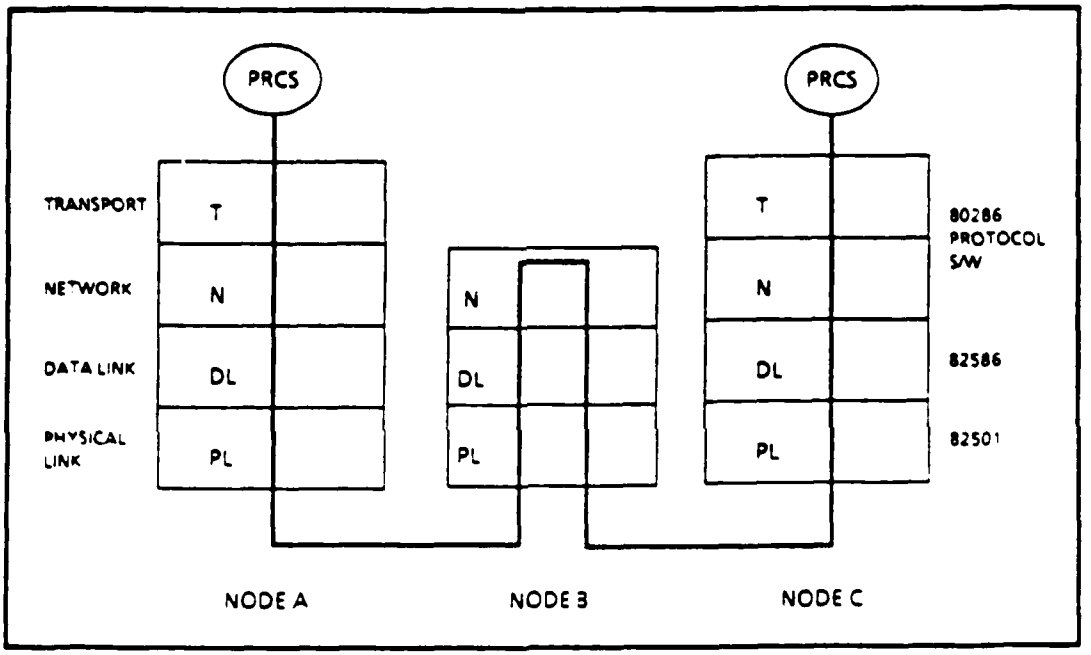


Figure 8. Routing Through Intervening Node(s) Source: [1, 3-7]

Layers three and four of the OSI are currently supported by the Node Operating System [1, 3-5]. The Node Operating System insures the interprocessor communication latency does not exceed the dimension of the hypercube [12, 164]. Figure 9 illustrates the cube's interconnection and bi-directional communications channel connection between nodes 0, 2, and 8. Each node of the iPSC is connected, via the i-th channel, to other nodes whose respective addresses differ in the i-th bit position [1, 3-3].

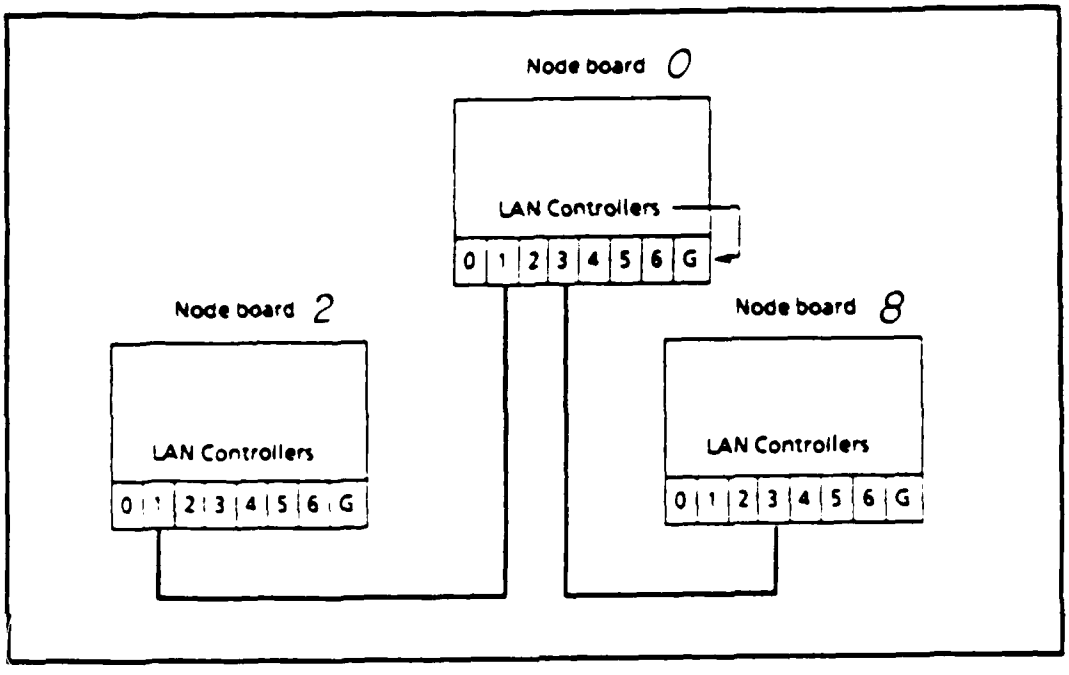


Figure 9. LAN Controller Interconnections Source: [1, 3-3]

Summary

This chapter presented an overview of concurrent systems architecture along with the iPSC's architecture. It has also presented a history of the development of the iPSC. Finally, the chapter illustrates how the Node Operating System and supporting hardware implements the cube interconnection function that defines the hypercube topology.

IV. Developed Routing Processes

This chapter develops the code used to implement the developed routing algorithm. The goal of the algorithm is to insure the routing process selects from among the shortest paths the one with the least delay. While it may inhibit selection of a path with a smaller delay time, the algorithm eliminates the possibility of looping in the network.

The developed code is comprised of two parts: a set of interface routines and the routing process. The interface routines, presented in the first section, enable the applications program to interface with the routing process. The routing process, presented in the second section, selects the next node routing for an applications message. The selection is based on a local routing table that is periodically updated. The source code for the routing process is listed in Appendix A, while Appendix B lists the source code for the interface routines.

All of the code was written in C and was designed for generic operation with minimal impact to the programmer. While the interface routines must be linked with the users node process(s), the routing process is a stand alone process loaded onto the iPSC's nodes by the host process. The applications programmer is restricted from using the process identification 32767, message types 32765 and 32767, and the maximum number of bytes in a message can not exceed 16372.

Interface Routines

The interface routines are divided into two subsets: one set for sending messages, and one set for receiving messages. Each set of interface routines is comprised of two routines that mirror the non-blocking and blocking versions of the sending and receiving routines found in the iPSC's library. The four interface routines are: **asendw**, **arecvw**, **asend**, and **arecv**. To simulate the current operation, the four

routines use the same argument list currently used by the iPSC's routines. Additionally, dynamic memory allocation is used in the routines to keep memory usage to a minimum.

The **asend** and **arecv** routines simulate the **send** and **recv** routines by using the iPSC **flick** routine to allow other processes to continue processing, so they are not true non-blocking routines. Except for the **flick** routine, the two send routines, as well as the two receive routines, operate identically to each other. Therefore, only the **asendw** and **arecvw** routines will be discussed in the following sections.

Send Routine. As stated above, the **asendw** routine uses the same argument list as the iPSC's **sendw** routine. The **asendw** routine sends messages of any data type allowed by the iPSC, and it maintains the iPSC's global send capability. The **asendw** routine consists of two parts: an initialization section and a section to select the iPSC routing or the adaptive routing.

The initialization portion is performed once (i.e., the first time the routine is called). It is used to determine the current node, the current process identification, and the current size of the routing algorithm overhead. The final step initializes an integer array with the current node's nearest neighbors.

Based on the destination node parameter, the second part of the routine determines if the iPSC routing should be used or if the adaptive routing should be selected. This decision is the first step in insuring that the fewest number of hops are performed. The routine compares the destination node of the message to the array of nearest neighbors. If a match occurs, the message is sent directly to its destination using the iPSC **sendw** routine. This technique eliminates two intranode hops. The iPSC **sendw** routine is also used directly, if the destination node is the current node or is negative (i.e., a global send). If no match occurs (i.e., the destination is more than one hop away) the message is passed to the adaptive routing process.

For the routing process to handle the messages properly additional data must be prefaced to the message. The overhead data includes, the destination node, the destination process identification, and the destination message type; along with the current node, the current process identification, and the original message length. The prefacing of the overhead data to the message enables the routing process to locate the original message parameters for use in its next node selection process.

The additional steps in the send routines include allocating memory to hold the message and the routing algorithm overhead, storing the overhead data in the allocated memory, and then appending the original message to the overhead data. This is followed by calling the iPSC **sendw** routine, freeing the allocated memory, and returning to the calling process.

Receive Routine. As stated above, the **arecvw** routine uses the same argument list as the iPSC's **recvw** routine. Like the **asendw** routine, the **arecvw** routine consists of two parts. The first section receives the message and the second section determines the originating process's parameters.

The first section begins with an initialization that establishes the number of bytes used by the overhead data. The initialization is followed by an allocation of memory to hold the overhead data and the expected message. The first section concludes with a call to the iPSC's **recvw** routine. The call is made using the variables from the calling process's argument list.

After the receive is completed, the second section of the routine determines if the message came from the routing process or an application process. This determination is required because of the overhead data added to the front of a message sent via the adaptive routing process, and is accomplished by checking the sending process identification parameter returned by **recvw**. If the message came from the routing process; the message length, sending node, and sending process identification parameters contained in the overhead data are assigned to the calling process's

arguments. Additionally, a temporary pointer is assigned to the beginning of the message in the allocated memory space. If the message was not sent by the routing process, no overhead data was added to the message, and the temporary pointer is assigned to the beginning of the allocated memory space. The **arecvw** routine concludes by performing a character by character copy of the received message to the calling process's message buffer, freeing the allocated memory, and returning to the calling process.

Routing Process

This section develops the routing process by defining the data structures and the algorithm used by the process. As discussed in Chapter II, three criteria must be met to establish a distributed adaptive routing process. They are a local means of determining the routing, a means of measuring the current traffic loads, and a means of informing other nodes of the routing information. This routing process meets these three criteria by means of a local routing table *del_times* and special messages (*DELAY_TYPE*) that are passed between nearest neighbor nodes identifying a nodes current routing information to the other nodes in the network. Additionally, the *DELAY_TYPE* message is used as a medium to determine the current traffic loads. this is detailed later.

Data Structures. The routing table is a two-dimensional array based on the current dimension of the iPSC. Each row of the array represents a possible destination node, while each column represents a nearest neighbor node. The i-th column of the table represents the i-th nearest neighbor node. The columns of the table are updated from information contained in the *DELAY_TYPE* messages that are passed between each of the nearest neighbor nodes and from the time taken by the *DELAY_TYPE* message to return to its originating node.

The routing process utilizes two vector arrays to update the routing table.

The first array *out_nodes* is an integer array containing the nearest neighbor nodes of the current node. The purpose of the array is to provide the column index for the routing table. The second array *start_times* is used to store the time the latest *DELAY_TYPE* message was transmitted to each neighbor. Both of these arrays are indexed, such that information pertaining to a node connected via the i-th communication channel is stored using the i-th index of the array.

The measured times used by the routing process are determined locally through the use of the iPSC **clock** routine. The value returned by the **clock** routine is only updated every 5 milliseconds [1, 3-24]. Since the **clock** routine in each of the nodes operates in a like manner and the measured times are relative times, the inaccuracy of the **clock** should not effect the operation of the routing process.

Routing Table Formation. The routing table is initialized to a maximum default time (msec). The local round trip delay time is determined by subtracting the time stored in the *start_times* array from the time the *DELAY_TYPE* message is received back from that node. The round trip delay times are stored in the routing table using the respective node and its index in the *out_nodes* array as the two indices. For example, in Node 0 the delay time for Node 1 is stored in element (1,0) of the routing table. When a neighbor node's *DELAY_TYPE* message is received, the data is added to the current round trip delay to update the routing table.

For example, the data in Figure 10 represents the routing table for a 3 dimension cube (i.e., 8 nodes) after Node 0 receives each of its nearest neighbor node's first *DELAY_TYPE* message followed by the return of its first own *DELAY_TYPE* message. For simplicity, it is assumed the maximum default time is 6000 msec and the round trip delay is 5 msec. The routing table starts with 6000 msec, to which an additional 6000 is added from the received *DELAY_TYPE* message, for a routing table of 12000 msec. The reception of the first round trip delay message results in the updating of the nearest neighbor elements with the measured delay information.

It should be noted that after Node 0's second *DELAY_TYPE* message is received by Node 2, Node 2's routing table will reflect that Nodes 1 and 4 could be reached via

		Neighbor Nodes		
		1	2	4
Destination Nodes	0	12000	12000	12000
	1	5	12000	12000
	2	12000	5	12000
	3	12000	12000	12000
	4	12000	12000	5
	5	12000	12000	12000
	6	12000	12000	12000
	7	12000	12000	12000

Figure 10. Example Routing Table for Node 0 of a 3 Dimension Cube - After First Update

Node 0 and that the round trip delay would be 10 msec. It should also be noted that, since the routing table reflects round trip times, the actual time is approximately one-half that shown in the table.

Continuing the updates, Figure 11 depicts the routing table of Node 0 after the completion of the second set of *DELAY_TYPE* messages returns, and all of the nearest neighbor messages are received.

Figure 12 depicts the routing table of Node 0 after the completion of the third iteration of *DELAY_TYPE* messages. It illustrates that after N update periods, where N is the cube dimension, each node's routing table contains a computed delay time for each possible destination node. Since the routing algorithm limits the number of hops to N, message routes greater than N will not be utilizied. The underlined entries in Figure 12 represent paths requiring more than N hops.

Although the paths will not be utilized, Figure 13 illustrates that after $N + 1$ delay times the entire routing table will be filled with delay times arrived at by link measurements.

		Neighbor Nodes		
		1	2	4
Destination Nodes	0	10	10	10
	1	5	12005	12005
	2	12005	5	12005
	3	10	10	12005
	4	12005	12005	5
	5	10	12005	10
	6	12005	10	10
	7	12005	12005	12005

Figure 11. Example Routing Table for Node 0 of a 3 Dimension Cube - After Second Update

		Neighbor Nodes		
		1	2	4
Destination Nodes	0	10	10	10
	1	5	15	15
	2	15	5	15
	3	10	10	12010
	4	15	15	5
	5	10	12010	10
	6	12010	10	10
	7	15	15	15

Figure 12. Example Routing Table for Node 0 of a 3 Dimension Cube - After Third Update

		Neighbor Nodes		
		1	2	4
Destination Nodes	0	10	10	10
	1	5	15	15
	2	15	5	15
	3	10	10	20
	4	15	15	5
	5	10	20	10
	6	20	10	10
	7	15	15	15

Figure 13. Example Routing Table for Node 0 of a 3 Dimension Cube - After Fourth Update

Routing Algorithm. The following section depicts the algorithm for the routing process. In particular, it details how the routing table is used to determine the next node on the shortest delay path to the destination node. Figure 14 depicts the routing process algorithm used in this research. The first section of the process determines the current node, opens a send and a receive channel, determines the nearest neighbor nodes of the current node, initializes the routing table to a default time value, and then transmits the first set of delay messages to the nearest neighbor nodes.

The remainder of the routing process is an infinite loop that repeatedly tests for the reception of a user application message, the reception of a routing process generated *DELAY_TYPE* message, and the time to periodically generate a new *DELAY_TYPE* message to update the routing table. As indicated in Figure 14, all user application messages are retransmitted before any *DELAY_TYPE* messages are handled. While this may result in messages going down a channel already known to be congested, it insures that passing application messages remains a higher priority than the processing of *DELAY_TYPE* messages.

Application Messages. When an application message is detected, it is stored in a temporary buffer. The overhead information stored in the first portion of the message is retrieved to identify the original destination information. If the message is destined for the current node or a nearest neighbor node the message is transmitted directly to its destination. Otherwise, the routing table is accessed to determine the next node.

Next Node Determination. The determination of the next node uses an *exclusive-or* operation on the current node and the destination node. The resulting binary value is checked, starting with the least significant bit. The i-th bit position of a detected "1" identifies the respective i-th communications channel is a valid message path. The first "1" bit detected establishes a temporary minimum

```
I. Initialization

    Open Channels for Sending/Receiving

    Determine Environment

        Current Node
        Current Cube Dimension
        Current Number of Nodes

    Initialize Tables

        Determine Nearest Neighbors
        Determine Start Times
        Determine Delay Times

II. Infinite Loop

    While an application message is waiting
        {
        Process routing overhead
        Determine next node
        Send Message
        }

    While a routing message is waiting
        {
        If message from local node
            Process round trip delay
        else
            {
            Retransmit message to sender
            Update routing Tables
            }
        }

    If it is time to send new routing message
        send message
    else
        flick process
```

Figure 14. Developed Routing Algorithm

delay time path and the next node on that path. As the remaining bits are tested, any detected "1's" are used to test for a new minimum delay time. If two or more nodes have the same delay time to the destination node, the node with the lowest index is used as the next node. After all the bits are checked, the node corresponding to the minimum delay time is used as the next node for the message. The *exclusive-or* operation insures the message path is limited to the fewest number of hops and as a result, eliminates the possibility of looping in the network.

Routing Messages. When an incoming message is a *DELAY_TYPE* message, the message is stored in a dedicated buffer. The sending node of the *DELAY_TYPE* message is used to determine the column of the routing table and the element of the other routing vectors. The element of the buffer corresponding to the current maximum number of nodes in the network is checked to determine if the message originated at the current node. If the message originated at the current node the round trip delay is calculated and the routing table updated with the round trip delay. Otherwise, the elements of the data buffer are added to the current round trip delay, and the sum used to update the respective column of the routing table.

Update Timing. Before checking for a new user application message, the node's clock is checked to determine if another set of delay messages need to be sent. Otherwise, the process **flicks** allowing the other process(s) to continue processing.

Summary

The code developed for this process was divided into two units, a main stand alone routing process and a set of interface routines to pass messages from a user's process to the routing process. It was designed to give each node the means to accomplish the three requirements for a distributed adaptive routing algorithm, stated in Chapter II. Through the use of the iPSC's **clock** routine and the *DELAY_TYPE* message the current node has the means to measure the current traffic load on its

channels. The current node has the means to determine which neighbor nodes are on the fewest number of hops path and can then select the next node on the path with the minimum delay time. Finally, using the *DELAY_TYPE* message, the adaptive routing algorithm has the means to pass its current routing information to other nodes in the network.

It should also be noted that this routing process suffers from the *bad news* problem as depicted in Tanenbaum. Once a routing path is established the news of its deterioration requires a number of update iterations to occur before the *bad news* is known by the remaining nodes in the network. Since the developed process is an initial effort in investigating the routing on the iPSC, the additional coding for correcting the *bad news* problem was left for a future enhancement.

V. Testing Methodology

This chapter explains the procedures used to validate and measure the performance of the adaptive routing process that was developed for this study. The testing of the routing process was accomplished in two phases. The first phase verified the routing process. This was achieved by insuring that predetermined message routes were used by the process and that the message contents were unaltered by the routing process. The second phase of the testing established the data necessary to compare the performance of the adaptive routing process with that of the current static routing process. In this phase, various network loading schemes were developed to test the routing algorithm with a range of loading message lengths and a number of communications bound paths.

The topology that was chosen for testing the network will be discussed in the first section of this chapter. The second section discusses the network loading factors used for the second phase of testing. The final section discusses the processes used to perform the testing, while the results of the second phase of testing are presented in Chapter VI.

Topology

For the adaptive routing algorithm to improve the performance of the network, three conditions must be occurring in the network. There must be communications occurring over multi-hop paths, there must be concurrent communications occurring over some portion of the multi-hop paths, and there must be sufficient additional communications bandwidth to offset the additional processing required by the adaptive routing process. Therefore, the first requirement for testing was to establish process(s) that provided the necessary communications.

A ring topology was chosen as the test topology. The ring was formed using the equation:

$$NN = (CN + 1) \bmod TN \tag{10}$$

where

NN = next node

CN = current node

TN = total number of nodes in the network.

This ring topology was chosen because it provided the necessary multi-hop paths and the routing was easily determined and verified. While the routing process can be used on any dimension cube, the testing was accomplished with the iPSC loaded as a dimension five hypercube. Table 1 depicts the number of hops required between the node pairs for the 32-node system. With this topology a total of 62 hops are

Table 1. Number of Hops for Each Node Pair

1	2	3	4	5
(0,1), (2,3)	(1,2)	(3,4)	(7,8)	(15,16)
(4,5), (6,7)	(5,6)	(11,12)	(23,24)	(31,0)
(8,9), (10,11)	(9,10)	(19,20)		
(12,13), (14,15)	(13,14)	(27,28)		
(16,17), (18,19)	(17,18)			
(20,21), (22,23)	(21,22)			
(24,25), (26,27)	(25,26)			
(28,29), (30,31)	(29,30)			

necessary for a message to pass around the ring.

The validation of the routing process was accomplished by appending the current node address to a message as it was sent around the ring. When the message returned to the initial node, it was sent to the host process for storage in a file and post-test analysis. The message also provided proof that the routing process did not overwrite or otherwise alter the contents of the message being passed.

Figure 15 depicts the nodes of the ring network as they are visited by a message being sent around the ring. The nodes represented on the inner ring are those nodes used by the static iPSC routing regardless of the network loading. The nodes on the outer ring represent nodes used during one of the test runs of the adaptive routing process. The test congested 16 of the 62 hops in the ring. The particular congested links are depicted by the double arcs in Figure 15. The method used to provide the congestion is described in the next section on network loading.

Node 0 was programmed as the ring controller. It received data from the Host Process specifying the length of the message and the number of times to pass the message around the ring. The multiple trips around the ring were used to offset the inaccuracy of the iPSC **clock** routine. Since the value returned by the routine is only updated every 5 milliseconds [1, 3-24], each message was send through the network 5 times to establish an average value. Upon receipt of the message from Node 31, Node 0 sent the message to the Host Process, which stored the data in an output file.

Network Loading

Communications loading for this research was provided by two processes. The code for the first process called the Ring Control Process is listed in Appendix D. It established the topology described above and part of the required multiple message load. The Ring Control Process created a set of variable length messages for each network loading test. The message length was varied to provide results that were not biased by any single message length. Table 2 lists the message lengths used by the Ring Control Process. The particular lengths were chosen to force transmission of full packets (1024 bytes) after the addition of the adaptive routing process overhead.

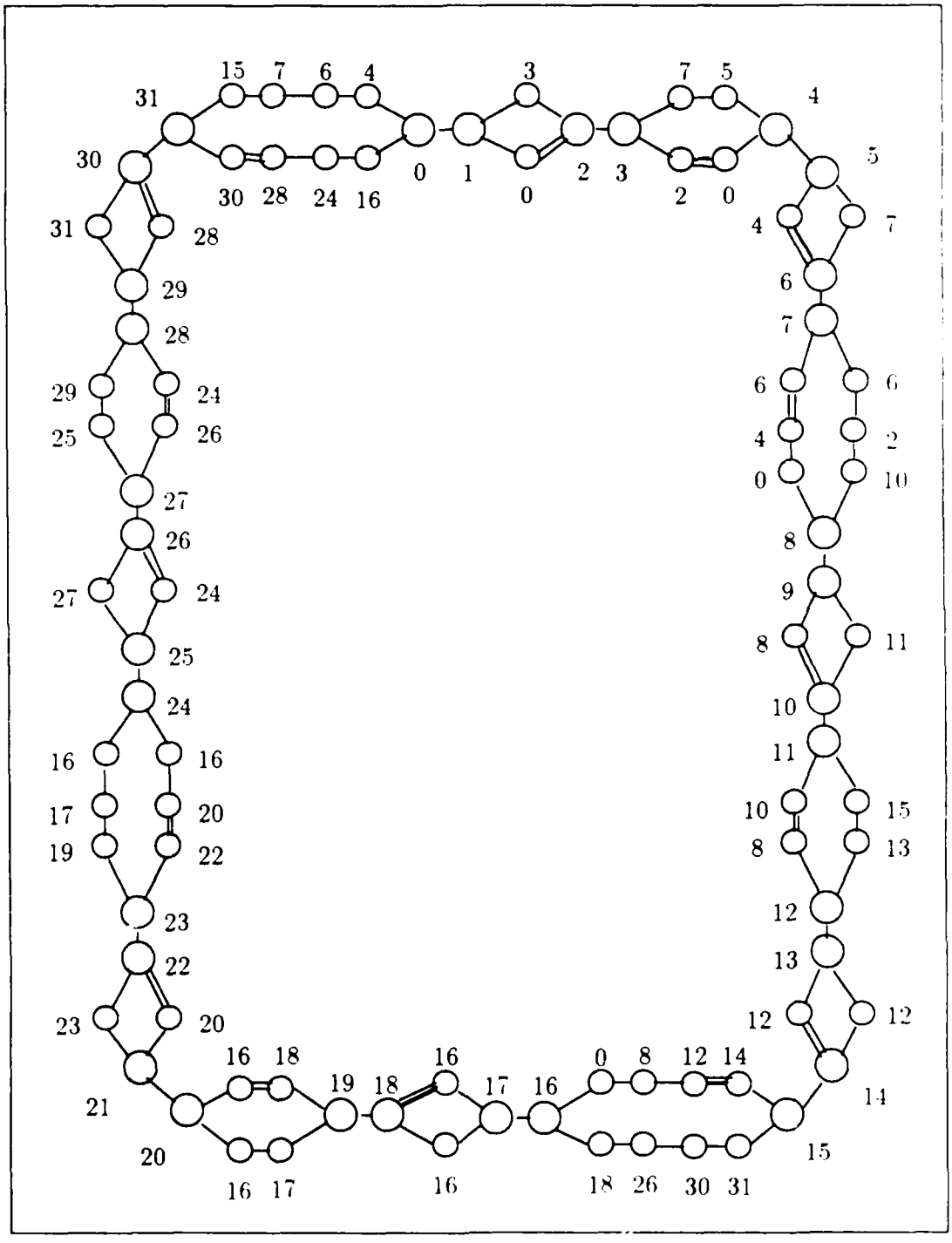


Figure 15. Example Ring Network with Intermediate Nodes

As stated earlier, the adaptive routing algorithm requires the presence of concurrent communications occurring over some portion of the multi-hop paths. The code for the second process called the Network Loading Process is listed in Appendix E. It provided the additional communications load on portions of the multi-hop paths used by the Ring Control Process. The Host Process passed to the Network Loading Process the desired number of congested links and the length of the message to be passed. Table 3 presents the number of congested links and the message lengths used by the Network Loading Process in the tests. Each of the different message lengths were used in testing each of the different number of congested links.

Table 2. Ring Message Length

Number of Bytes/Msg.
0
2036
4084
6132
8180
10228
12276
14324
16372

Table 3. Network Loads

Message Lengths (Bytes)	Number of Congested Links
0	0
4096	4
8192	8
12288	12
16384	16

The number of congested links was the value used by the Network Loading Process to determine which nodes would initially send a message and which nodes would initially wait to receive a message. Table 4 presents the links congested for the various numbers. After the initial send/receive, each process called the opposite

Table 4. Congested Links

Number of Congested Links				Congested Link Pairs
			4	(0,2) (1,3) (4,6) (5,7)
		8		(8,10) (9,11) (12,14) (13,15)
	12			(16,18) (17,19) (20,22) (21,23)
16				(24,26) (25,27) (28,30) (29,31)

function and in a ping-pong fashion the send/receive sequence was repeated. Nodes not involved in sending or receiving used the iPSC **flick** routine to cancel their processing time slice.

Comparison Metric. A complete test of the routing process consisted of the Ring Control Process sending each of the message lengths, listed in Table 2, five times for each of the 21 combinations of message lengths and number of congested links as given in Table 3. The time required by the Ring Control Process to send a message around the ring formed the basis of the data collected during the testing.

Each combination of message length and links congested by the Network Loading Process was represented by the summation of the average times of each message length to be passed by the Ring Control Process.

Configurations Tested

Three configurations of processes were tested to measure and compare the performance of the adaptive routing process with the current routing process. Table 5 lists the configurations. Each of the configurations used the same Host Process and the same Network Loading Process. Only the Routing Process and the Ring Control Process were altered. The Routing Process was altered to perform adaptive or static routing. The Ring Control Process was altered to interface with the Routing Process or the current iPSC message passing routines. In each configuration, three processes were running on each node of the iPSC. The results of the testing are compared in Chapter VI to determine the effect of the adaptive routing algorithm.

Table 5. Test Configurations

Configuration	Description
1	Distributed Adaptive Routing (DAR)
2	Simulated Static Routing (SSR)
3	Current Static Routing with Added Process (CSR/AP)

The Distributed Adaptive Routing (DAR) configuration was comprised of the Routing Process to route the messages according to its routing table, and the Ring Control Process linked with the Routing Process's interface routines. The Simulated Static Routing (SSR) configuration used a modified version of the Routing Process to intercept messages and route them according to the iPSC static routing, and the Ring Control Process again linked with the Routing Process's interface routines. Because

the adaptive routing process was implemented at the applications layer of the OSI protocol and not the network layer, the simulation of the iPSC routing process was necessary to isolate the effects of the adaptive routing process. Therefore, the two configurations can be compared to determine the effect of the adaptive routing over the static routing.

The Current Static Routing with Added Process (CSR/AP) configuration used the iPSC **sendw** and **recvw** routines in the Ring Control Process instead of the adaptive routing process's interfaces **asendw** and **arecvw**. Since the Ring Control Process was not linked with the Routing Process's interface routines the Routing Process could not intercept the message traffic. Because the other two configurations consisted of three processes, the Routing Process was also loaded in this comfiguration to keep the process loading equal. This third configuration can be compared with the DAR configuration to establish the processing overhead caused by the adaptive routing process.

Summary

This chapter has presented the testing methodology used in this research, to include how the metric (delay time) used in the study was established. It also presented the topology and the various network loads used to establish the communications load. This was followed by the various configurations developed to test and compare the adaptive routing algorithm against the current iPSC routing performance. The additional processes are listed in the appendices; Appendix C - the Host Process, Appendix D - the Ring Control Process, and Appendix E - the Network Loading Process. The Makefile used to compile and link the processes is listed in Appendix F. Chapter VI presents the results and conclusions from the testing.

VI. Conclusions

The purpose of this study was to examine the effect of a distributed adaptive routing algorithm on a concurrent class computer. The focus of the study involved the comparison of an implemented adaptive routing algorithm, a simulation of the current static routing algorithm and the current static routing algorithm with an added process for load balancing. This chapter compares results obtained from the testing methodology described in the Chapter V, and concludes with recommendations for additional study.

Results

Each of the three configurations described in Chapter V was tested using the twenty different message loads listed in Table 3. In addition, a test of the three configurations was performed without any additional message traffic from the Network Loading Process. Again, the three configurations were the Distributive Adaptive Routing (DAR), the Simulated Static Routing (SSR), and the Current Static Routing with Added Process (CSR/AP).

Table 6 is the summation of the average delay times for the nine control message lengths, of Table 2. Again the summation was done to measure the performance of the adaptive routing algorithm unbaised by a fixed message length. Based on the data formulated from the average delay times of the various traffic loads, five graphs, Figures 16- 20, were developed to compare the message delay times of the three configurations.

The data in Table 6 indicates that for the network loads tested the adaptive routing process was from 19.1–95.6 secs faster than the simulated iPSC static routing process for the test cases used. Additionally, the current static routing was 69.5–90.46 secs faster than the adaptive routing. Based on the data from the testing performed with no loading messages, the overhead caused by implementing the

Table 6. Summarized Timing Measurements (in Seconds)

Conf.	Number of Bytes/Msg.	Number of Congested Links				
		0	4	8	12	16
1 DAR	0	78.83	116.71	151.21	182.45	219.37
	4096	–	103.04	136.56	163.83	201.10
	8192	–	101.81	134.87	159.81	202.84
	12288	–	101.86	135.60	163.47	202.92
	16384	–	103.05	133.56	161.98	200.75
2 SSR	0	76.56	134.95	191.94	251.68	298.61
	4096	–	120.47	187.94	252.82	302.37
	8192	–	125.49	179.96	248.15	296.02
	12288	–	120.20	179.11	248.16	307.42
	16384	–	120.99	180.56	252.41	300.77
3 CSR/AP	0	5.55	46.26	81.66	128.31	169.27
	4096	–	30.65	55.53	80.67	101.65
	8192	–	34.67	56.49	76.58	102.23
	12288	–	34.24	55.06	79.84	103.13
	16384	–	33.08	51.79	83.51	98.42

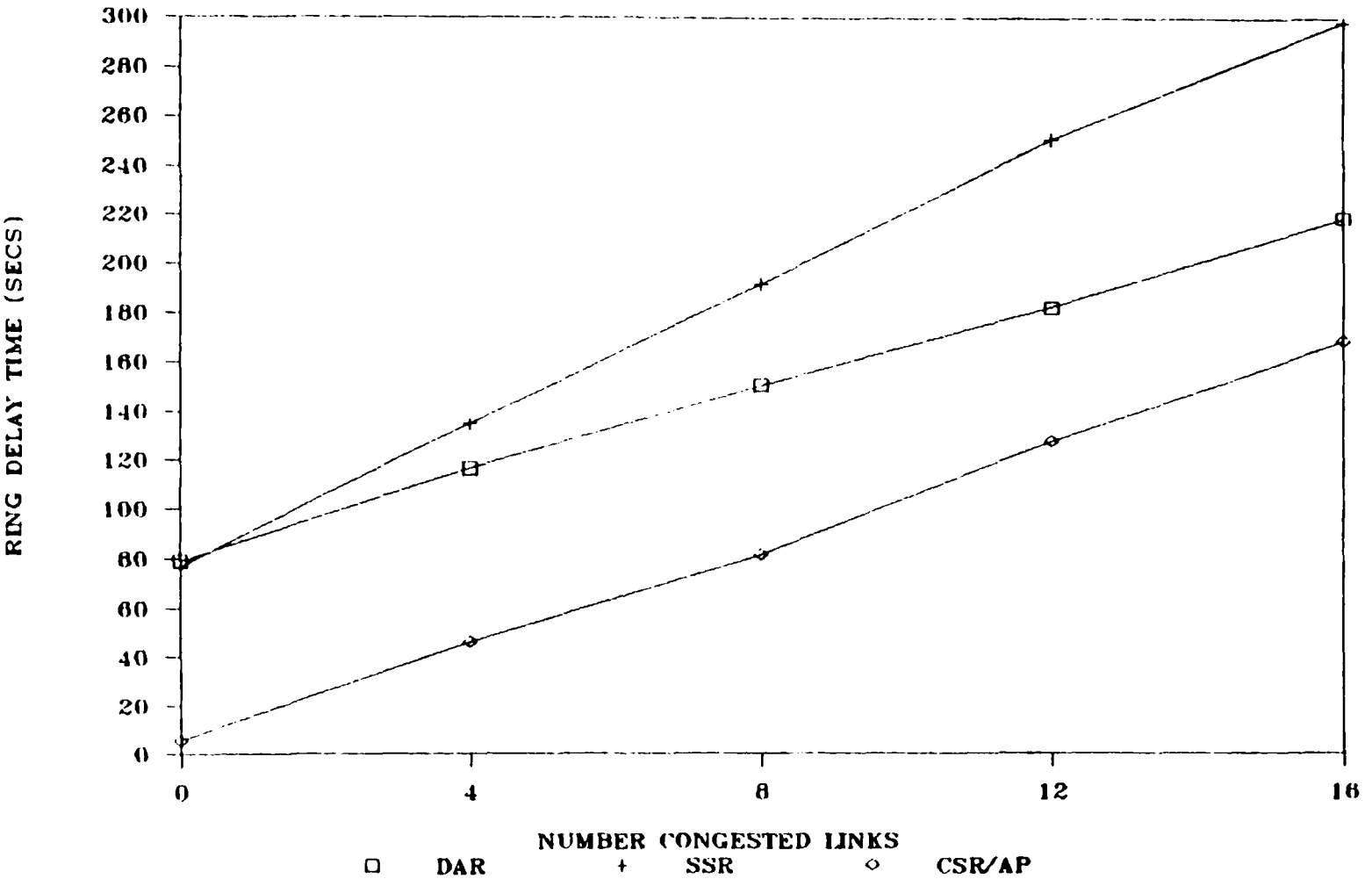


Figure 16. 0-Byte Loading Message

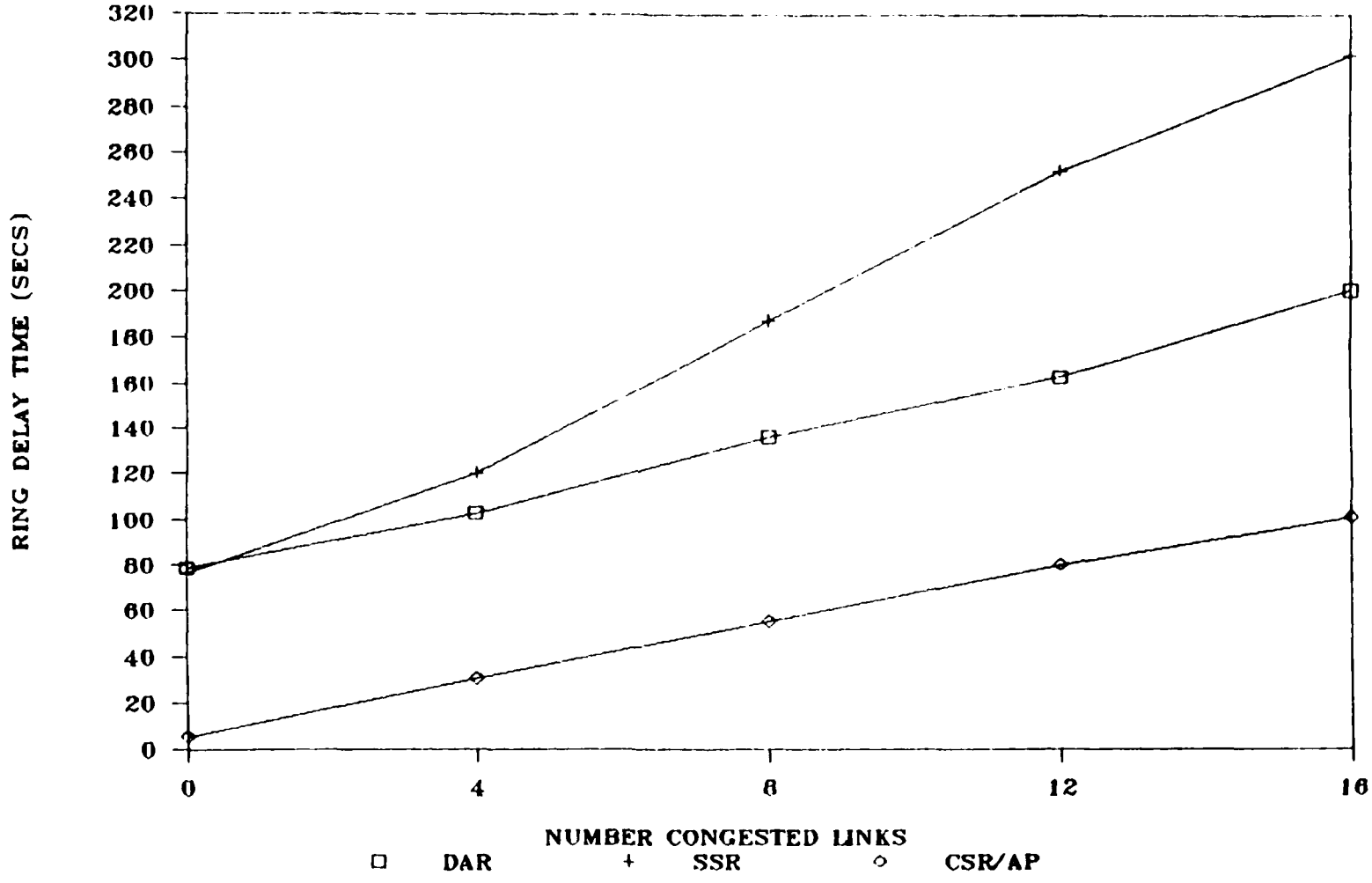


Figure 17. 4096-Byte Loading Message

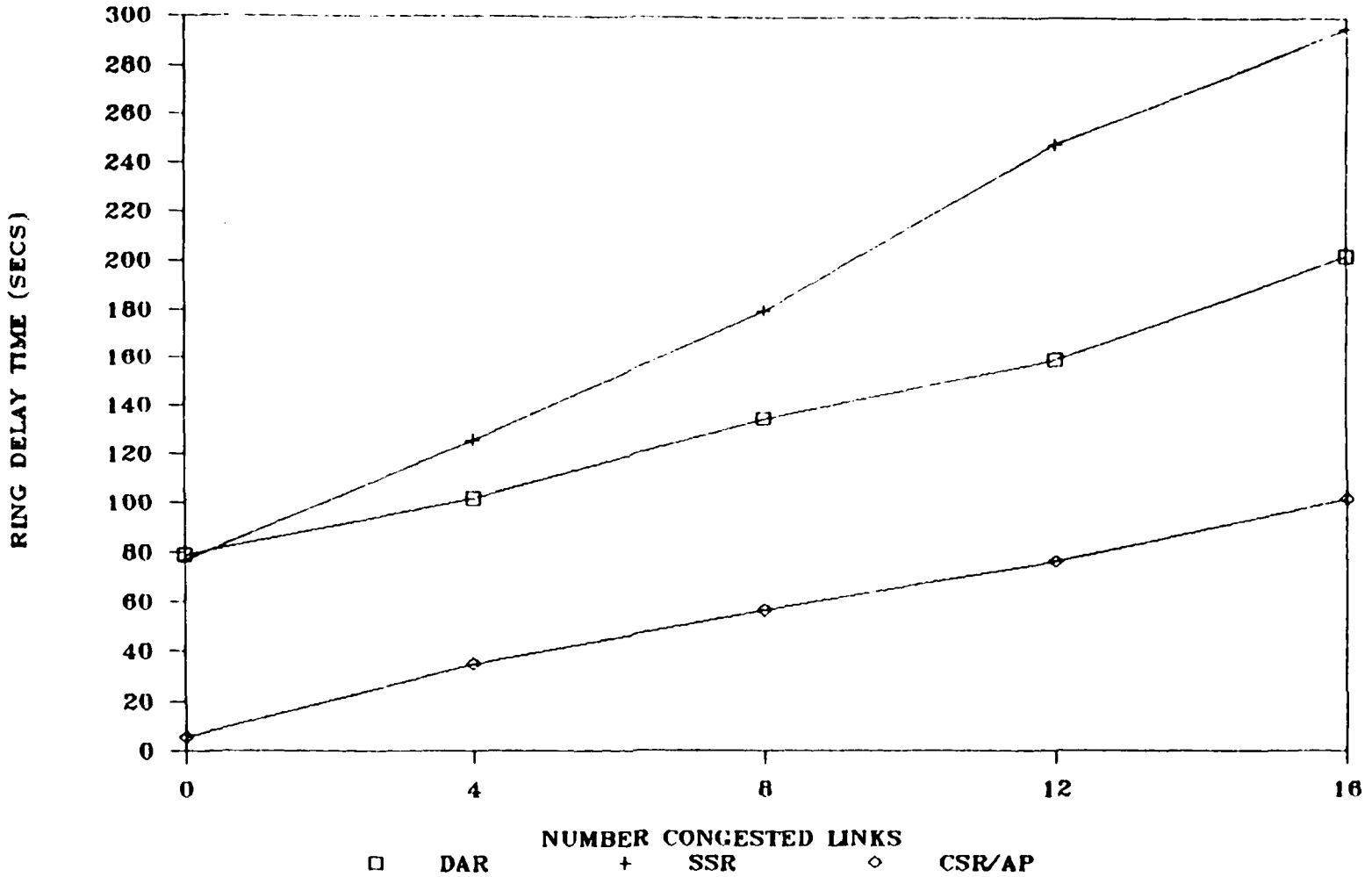


Figure 18. 8192-Byte Loading Message

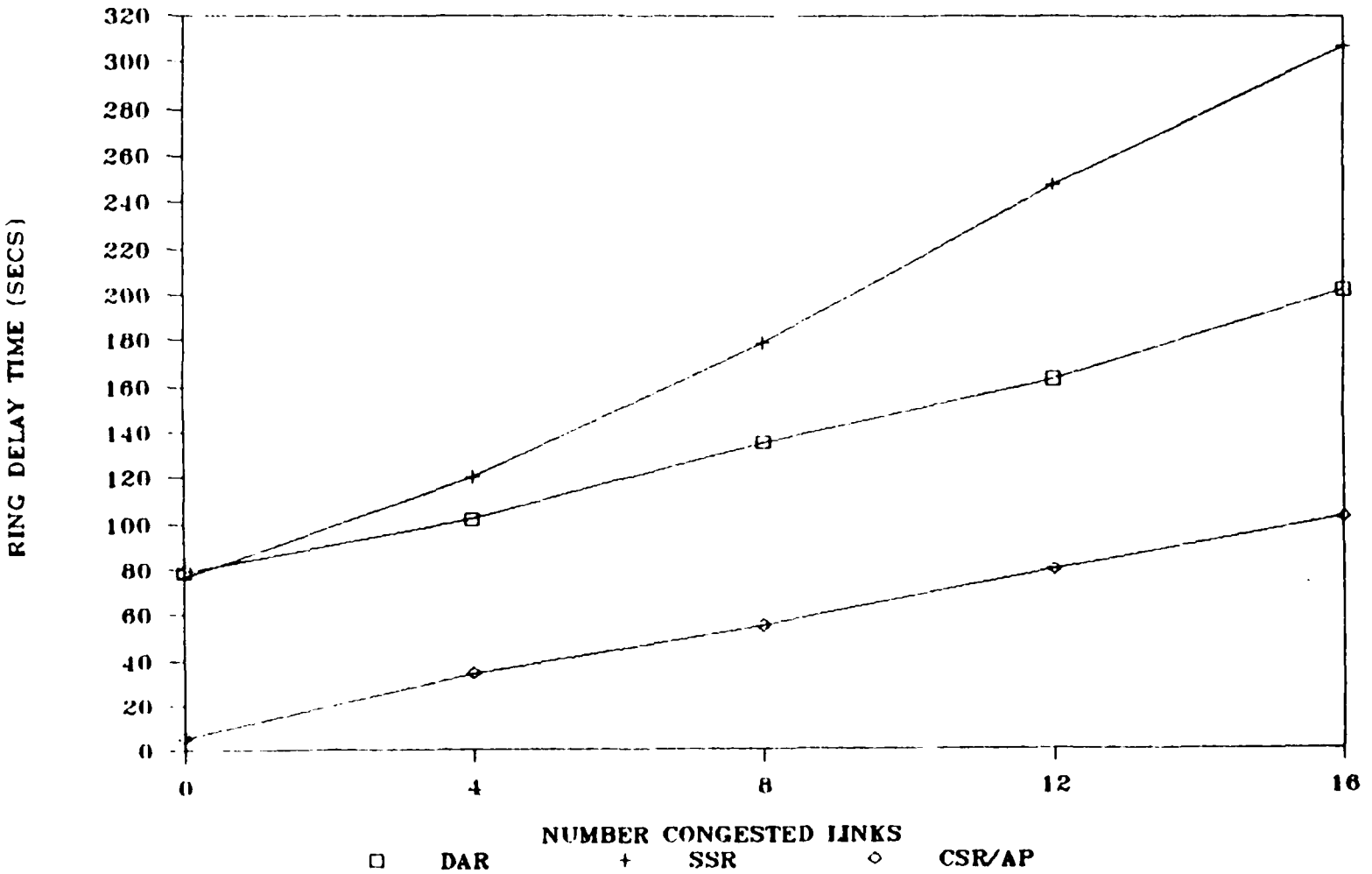


Figure 19. 12288-Byte Loading Message

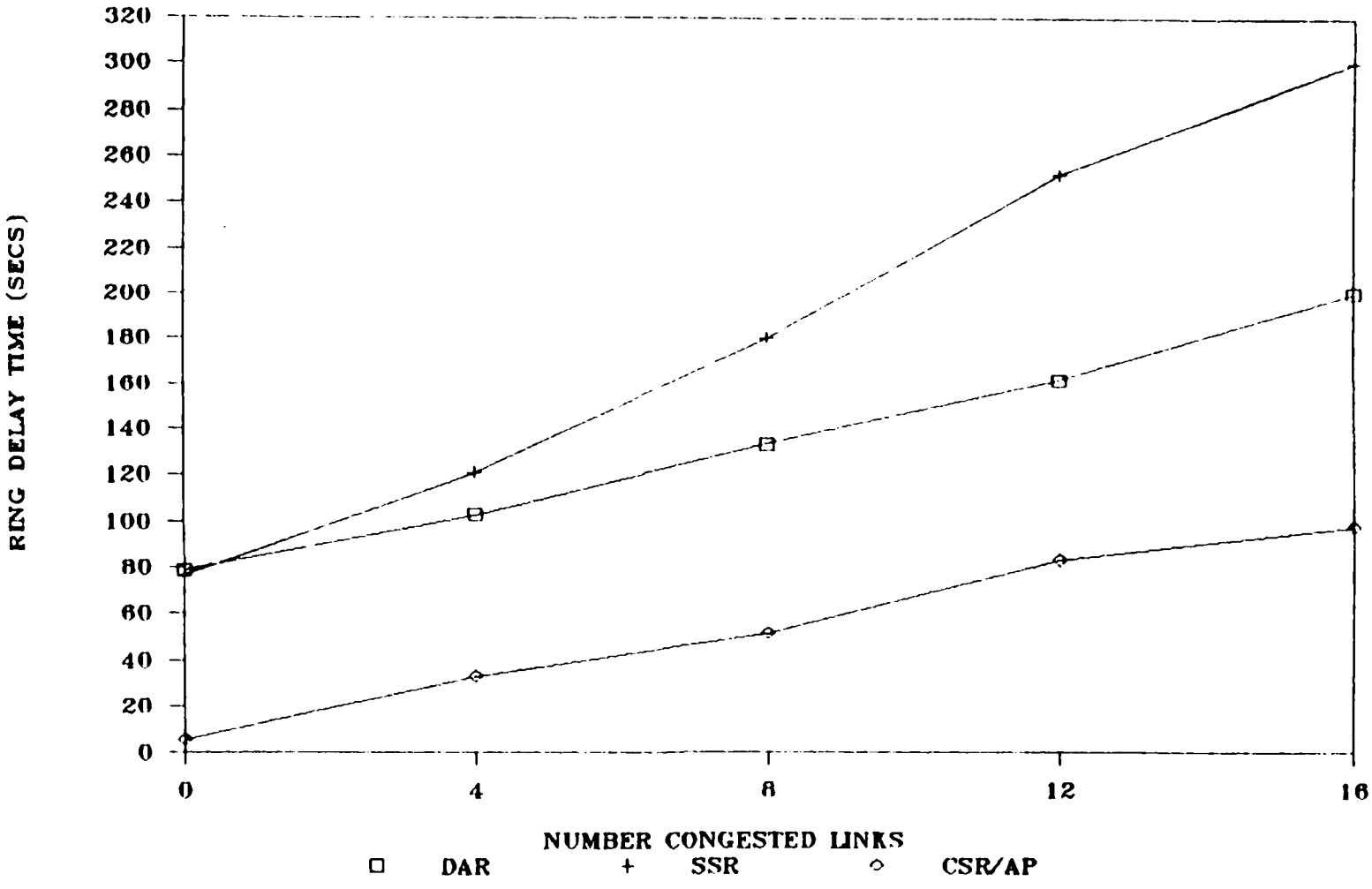


Figure 20. 16384-Byte Loading Message

routing algorihtm at the applications layer was 71 secs. It should be noted that the current static routing process was from 88.64-186.1 secs faster than the simulated static routing.

Analysis of the five figures results in the following conclusions. Comparing configurations one (DAR) and two (SSR) indicates the adaptive routing process, as expected, has a smaller delay time than the SSR configuration. This is due to the additional communication channels available to the DAR configuration. The figures also reveal that the length of the congesting message did not have an impact on the ring delay time, but the number of congested links did. This indicates that the links were congested equally, regardless of the actual congesting message length.

It was anticipated that at a certain level of congestion the DAR confguration would also have a smaller delay than the configuration three (CSR/AP). Unfortunately, this did not occur. Post-test analysis of the Network Loading Process indicates that the link congestion was not the overriding communications factor, but that the time-slicing of the different processes was the biggest factor in the delay times of the messages. Not only did the time-slicing impede the messages, it also decreased the arrival rate of messages into the network from the Network Loading Process, therefore, decreasing the communications load on the network.

Future Research

While the testing accomplished in this study attempted to cover a wide range of possible communications loads, additional testing should be performed. More research into how the iPSC sends messages when more than one process is operating, as well as, expanded testing with altered congesting techniques. Along with the expanded testing of the message passing system, research into the operating system itself could yield a better understanding of the operation of the iPSC. Additional study should also involve implementing the developed adaptive routing algorithm developed at the network layer of the OSI protocol.

Appendix A. *Routing Process*

```
/*************************************************************
*                                                            *
*  DATE:  09/20/87                                           *
*  VERSION:  2.2                                             *
*                                                            *
*  TITLE:  Simulated Distributed Adaptive Routing Process    *
*  FILENAME:  route.c                                        *
*  PROJECT:  Thesis                                          *
*  OPERATING SYSTEM:  XENIX                                  *
*  LANGUAGE: C                                               *
*  FILE PROCESSING: compiled with options -ALfu              *
*  FUNCTION:  This process provides an alternate routing     *
*  capability for the iPSC.  It provides selection of the    *
*  next node by shortest path and smallest delay to the      *
*  destination node.  If the destination node is an          *
*  adjacent node, the message is sent directly to            *
*  destination process.                                      *
*                                                            *
*************************************************************/
#include "/usr/ipsc/lib/cnode.def"

#define MAX_CUBEDIM     7
#define MAX_NUM_NODES   128

#define ROUTE_TYPE 32767
#define ROUTE_PID  32767

#define DELAY_TYPE 32765
#define DELAY_SIZE (4*(MAX_NUM_NODES+1))

#define MAX_TIME   6000
        /*  Time in msecs to identify bad link. */
#define ADJ_TIME   2500
        /*  Time in msecs to recalculate delay. */
#define BAD_NODE   3333
        /*  Used as a default bad node number.  */
#define ROUTE_TEST   0
        /*  Used to collect msg path data.      */
```

```
#define MSG_SIZE   16384
        /*  Maximum number bytes in a message.  */

/************************
*  Program variables:  *
************************/
int i,j;
int send_chan, recv_chan;
int my_node, next_node;
int cube_dim, num_nodes;
int d_node, d_pid, d_type;
int rcnt, rnode, rpid;
int adj_time;

int  out_nodes[MAX_CUBEDIM];
long update_time;
long del_times[MAX_NUM_NODES][MAX_CUBEDIM];
long start_times[MAX_CUBEDIM];
long delay_buf[MAX_NUM_NODES + 1];

int recv_buf[MSG_SIZE/2];
int *temp1_ptr;
char *temp_str;

/*****************************************************************/
main()
{
   /************************************
   *  Open channels for communicating   *
   *  with the other nodes.             *
   ************************************/
   send_chan = copen(ROUTE_PID);
   recv_chan = copen(ROUTE_PID);

   /************************************
   *  Each process identifies its node  *
   *  and determines adjacent nodes.    *
   ************************************/
   my_node   = mynode();
   cube_dim  = cubedim();
   num_nodes = 1<<cube_dim;
   adj_time  = ADJ_TIME + my_node;
```

```
/************************************
*  Initialize the routing table and  *
*  and send first set of delay msgs  *
************************************/
init_tables(my_node,cube_dim,num_nodes);
update_time = clock();
get_out_times(send_chan,my_node,cube_dim,num_nodes);

/************************************
*     BEGIN ROUTE CODE:              *
************************************/
for (;;)
   {

   /*****************************************************
   *  This section controls the passing of application  *
   *  messages to the desired destination node/pid.     *
   *****************************************************/
   while (probe(recv_chan,ROUTE_TYPE) >= 0)
      {
      recvw(recv_chan, ROUTE_TYPE, recv_buf, MSG_SIZE,
         &rcnt, &rnode, &rpid);

      temp1_ptr =  recv_buf;
      d_node    = *temp1_ptr++;
      d_pid     = *temp1_ptr++;
      d_type    = *temp1_ptr;

      /*************************************************
      *  Used to id nodes traversed in path testing.     *
      *  Not used in time testing.                       *
      *************************************************/
      #if (ROUTE_TEST)
         {
         for(i=0; recv_buf[i] >= 0; i++)
            ;
         recv_buf[i] = my_node;
         }
      #endif

      j = node_index(d_node,cube_dim);
```

```
   if ((j < cube_dim) || (d_node == my_node))
       sendw(send_chan, d_type, recv_buf,
                              rcnt, d_node, d_pid);

   else
      {
      next_node = n_node(my_node,d_node,cube_dim);
      sendw(send_chan, ROUTE_TYPE, recv_buf,
         rcnt, next_node, ROUTE_PID);
      }

   }

/*********************************************************
*  This section controls passing of network delay      *
*  information that is used to determine least         *
*  delay path to destination node.                     *
*********************************************************/
while (probe(recv_chan,DELAY_TYPE) >= 0)
   {
   recvw(recv_chan, DELAY_TYPE, delay_buf, DELAY_SIZE,
      &rcnt, &rnode, &rpid);

   /*****************************************
   * Get channel index for del_times array. *
   *****************************************/
   j = node_index(rnode,cube_dim);

   if (j < cube_dim)
      {
      if (delay_buf[num_nodes] == my_node)
         {
         del_times[rnode][j] =
            clock() - start_times[j];
         start_times[j] = 0;
         }
      else
         {
         sendw(send_chan, DELAY_TYPE, delay_buf,
            rcnt, rnode, rpid);
```

```
                for(i=0; i < num_nodes; i++)
                   del_times[i][j] =
                      delay_buf[i] + del_times[rnode][j];
                }
             }
          else
             {
             sprintf(temp_str,
                "Rec'd delay msg from invalid node %d",rnode);
             syslog(ROUTE_PID,temp_str);
             }

          }

       /*******************************************************
       *  This statement controls the frequency of           *
       *  updates to the routing table.                      *
       *******************************************************/
       if (clock() - update_time > adj_time)
          {
          update_time = clock();
          get_out_times(send_chan,my_node,cube_dim,num_nodes);

          }
       else
          flick();

       }     /*   End of infinite for-loop     */
}      /*    END OF MAIN FUNCTION    */

/************************************************
*    BEGINNING OF FUNCTION DEFINITION SECTION    *
************************************************/

/*************************************************************
*  Function to initialize the routing table arrays:          *
*  out_nodes[], start_times[], and del_times[][].            *
*************************************************************/
init_tables(my_node,cube_dim,num_nodes)
int my_node;
int cube_dim;
int num_nodes;
```

```
{
   int i, j;

   for( i=0; i < cube_dim; i++)
      {
      j = 1 << i;
      out_nodes[i]   = my_node ^ j;
      start_times[i] = 0;
      }

   for( i=0; i < num_nodes; i++)
      for( j=0; j < cube_dim; j++)
         del_times[i][j] = MAX_TIME;
}  /*  End of init_tables function   */


/*************************************************************
*  Function node_index is used to return the index into      *
*  the del_times and out_nodes arrays.                       *
*************************************************************/
int node_index(test_node,cube_dim)
int test_node;
int cube_dim;
{
   int j;

   for( j=0; test_node != out_nodes[j] && j < cube_dim; j++)
      ;

   return(j);
}   /*  End of node_index function   */

/*************************************************************
*  Function get_out_times used to update the delay times.    *
*  It resets start_times and sends the DELAY messages        *
*  to the neighboring nodes.                                 *
*************************************************************/
int get_out_times(ci,my_node,cube_dim,num_nodes)
int ci, my_node;
int cube_dim, num_nodes;
{
   int i;
```

```
   get_delay_buf(my_node,cube_dim,num_nodes);

   for ( i=0; i < cube_dim; i++)
      if (start_times[i] == 0)
         {
         start_times[i] = clock();
         sendw(ci, DELAY_TYPE, delay_buf,
            DELAY_SIZE, out_nodes[i], ROUTE_PID);
         }
}     /*   End of get_out_times function   */

/*************************************************************
*  Function get_delay_buf is used to retreive the data      *
*  necessary information to build the delay_buf message.    *
*************************************************************/
int get_delay_buf(my_node,cube_dim,num_nodes)
int my_node, cube_dim, num_nodes;
{
   int i, j, adj_time;
   long min;

   for( j=0; j < cube_dim; j++)
      if (start_times[j] != 0)
         {
         adj_time = ADJ_TIME + my_node;
         for( i=0; i < num_nodes; i++)
            del_times[i][j] += adj_time;
         }

   for( i=0; i < num_nodes; i++)
      {
      min = MAX_TIME;
      for( j=0; j < cube_dim; j++)
         if (del_times[i][j] < min)
            min = del_times[i][j];
      delay_buf[i] = min;
      }

   delay_buf[i] = my_node;
   delay_buf[my_node] = 0;
}   /*   End of get_delay_buf function    */
```

```
/*************************************************************
*  Function n_node determines the next node in path          *
*  to destination or assigns a default node.                 *
*  If variable out_node isn't reassigned, the                *
*  original destination is returned as the next node.        *
*************************************************************/
int n_node(curr_node,dest_node,cube_dim)
int curr_node;
int dest_node;
int cube_dim;
{
   int i = 0;
   int j = 1;
   int temp_node, out_node;
   long min;

   /*****************************************
   *  Establish time and default node      *
   *  used to determine a congested link.  *
   *****************************************/
   min = MAX_TIME + 1;
   out_node  = BAD_NODE;

   temp_node = curr_node ^ dest_node;

   for ( i=0; i < cube_dim; i++)
      {
      if ( (temp_node & j) &&
         (min > del_times[dest_node][i]))
            {
            min = del_times[dest_node][i];
            out_node = out_nodes[i];
            }
      j <<= 1;
      }

   return((out_node != BAD_NODE) ? out_node : dest_node);
}  /*    End of n_node function     */


/*    END OF FUNCTION DEFINITION SECTION    */
```

Appendix B. *Interface Functions*

```
/*************************************************************
*                                                            *
*  DATE:  09/20/87                                           *
*  VERSION:  2.2                                             *
*                                                            *
*  TITLE:  Blocking Send for Adaptive Routing Process        *
*  FILENAME:  asendw.c                                       *
*  PROJECT:  Thesis                                          *
*  OPERATING SYSTEM:  XENIX                                  *
*  LANGUAGE: C                                               *
*  FILE PROCESSING: compiled with options -ALfu              *
*  FUNCTION:  This fuction can be linked to any program      *
*  using the follwing parameters:                            *
*     int ci;                                                *
*     int type;                                              *
*     char buf[n];                                           *
*     int len;                                               *
*     int node;                                              *
*     int pid;                                               *
*                                                            *
*************************************************************/

#define ROUTE_TYPE 32767
#define ROUTE_PID  32767

#define MAX_CUBEDIM    7
#define OVR_HEAD       6

asendw(ci, type, buf_ptr, len, node, pid)

int ci;         /* channel value for message sending       */
int type;       /* type value of the message               */
char *buf_ptr; /* pointer to buffer holding message        */
int len;        /* length value in bytes being send        */
int node;       /* value of the destination node           */
int pid;        /* value of the destination process id     */
{
   int *malloc();
   void free();
```

```
int  *send_buf_ptr;
int  *ovr_head_ptr;
char *temp1_ptr;
char *temp2_ptr;
int i, j;
int out_len;

static int initialized    = 0;
static int my_node, my_pid, cube_dim;
static int over_head_size;
static int nearest_nodes[MAX_CUBEDIM];

if (!initialized)
   {
   my_node  = mynode();
   my_pid   = mypid();
   cube_dim = cubedim();
   over_head_size = OVR_HEAD * sizeof(int);

   for (i=0; i < cube_dim; i++)
      {
      j = 1 << i;
      nearest_nodes[i] = my_node ^ j;
      }
   initialized = 1;
   }

/*********************************************************
*  Code to allow direct transmission of a message to  *
*  itself, a neighbor node, or make a global          *
*  transmission without using the routing routine.    *
*********************************************************/
for(i=0; node != nearest_nodes[i] && i < cube_dim; i++)
   ;

if(node == my_node || node < 0 || i < cube_dim)
   {
   sendw(ci, type, buf_ptr, len, node, pid);
   }
else
   {
```

```
        /*************************************************
        *  This portion of the code prepares the message   *
        *  overhead so the message can be properly         *
        *  handled by the routing process.  The routing    *
        *  overhead is stored in first six integers of     *
        *  message buffer.                                 *
        *************************************************/
        out_len      = over_head_size + len;
        ovr_head_ptr = malloc(out_len);
        send_buf_ptr = ovr_head_ptr;

        *ovr_head_ptr++  = node;
        *ovr_head_ptr++  = pid;
        *ovr_head_ptr++  = type;
        *ovr_head_ptr++  = my_node;
        *ovr_head_ptr++  = my_pid;
        *ovr_head_ptr++  = len;

        /***********************************************
        *  Code to copy message into new data buffer.  *
        ***********************************************/
        temp1_ptr = (char *)ovr_head_ptr;
        temp2_ptr = buf_ptr;

        for(i=0; i < len; i++)
           *temp1_ptr++ = *temp2_ptr++;

        sendw(ci, ROUTE_TYPE, send_buf_ptr,
           out_len, my_node, ROUTE_PID);

        free(send_buf_ptr);
        }

    }  /* End of asendw function   */
```

```
/*************************************************************
*                                                            *
*  DATE:  09/20/87                                           *
*  VERSION:  2.2                                             *
*                                                            *
*  TITLE:  Blocking Receive for Adaptive Routing Process     *
*  FILENAME:  arecvw.c                                       *
*  PROJECT:  Thesis                                          *
*  OPERATING SYSTEM:  XENIX                                  *
*  LANGUAGE: C                                               *
*  FILE PROCESSING: compiled with options -ALfu              *
*  FUNCTION:  This fuction can be linked to any program      *
*  using the follwing parameters:                            *
*     int ci;                                                *
*     int type;                                              *
*     char buf[n];                                           *
*     int len;                                               *
*     int cnt;                                               *
*     int node;                                              *
*     int pid;                                               *
*                                                            *
*************************************************************/

#define ROUTE_PID  32767
#define ROUTE_TYPE 32767

#define OVR_HEAD  6

arecvw(ci, type, buf_ptr, len, pt_cnt, pt_node, pt_pid)

int ci;         /* channel value from calling programming   */
int type;       /* type value from calling programming      */
char *buf_ptr; /* pointer to calling program buffer space  */
int len;        /* length value of buffer from calling prg  */
int *pt_cnt;    /* pointer to count of rec'd message        */
int *pt_node;   /* pointer to source node of rec'd message  */
int *pt_pid;    /* pointer to source proc. id of rec'd mesg */

{
   int *malloc();
   void free();
```

```
        int i, buf_size;
        int *ovr_head_ptr;
        int *recv_buf_ptr;

        char *temp1_ptr;
        char *temp2_ptr;

        static int initialized = 0;
        static int over_head_size;
        if (!initialized)
           {
           over_head_size = OVR_HEAD * sizeof(int);
           initialized    = 1;
           }

        buf_size     = over_head_size + len;
        recv_buf_ptr = malloc(buf_size);

        recvw(ci, type, recv_buf_ptr, buf_size,
           pt_cnt, pt_node, pt_pid);

        if (*pt_pid == ROUTE_PID)
           {
           ovr_head_ptr = recv_buf_ptr + 3;

           *pt_node = *ovr_head_ptr++;
           *pt_pid  = *ovr_head_ptr++;
           *pt_cnt  = *ovr_head_ptr++;
           }

        else
           ovr_head_ptr = recv_buf_ptr;

        temp1_ptr = (char *)ovr_head_ptr;
        temp2_ptr = buf_ptr;

        for(i=0; i < *pt_cnt; i++)
           *temp2_ptr++ = *temp1_ptr++;

        free(recv_buf_ptr);
        return;
     }  /* End of arecvw function  */
```

```
/*****************************************************************
*                                                                *
*  DATE:  09/20/87                                               *
*  VERSION:  2.2                                                 *
*                                                                *
*  TITLE:  Non-Blocking Send for Adaptive Routing Process        *
*  FILENAME:  asend.c                                            *
*  PROJECT:  Thesis                                              *
*  OPERATING SYSTEM:  XENIX                                      *
*  LANGUAGE: C                                                   *
*  FILE PROCESSING: compiled with options -ALfu                  *
*  FUNCTION:  This fuction can be linked to any program          *
*  using the follwing parameters:                                *
*     int ci;                                                    *
*     int type;                                                  *
*     char buf[n];                                               *
*     int len;                                                   *
*     int node;                                                  *
*     int pid;                                                   *
*                                                                *
*****************************************************************/

#define ROUTE_TYPE 32767
#define ROUTE_PID  32767

#define MAX_CUBEDIM    7
#define OVR_HEAD       6

asend(ci, type, buf_ptr, len, node, pid)

int ci;         /* channel value for message sending       */
int type;       /* type value of the message               */
char *buf_ptr; /* pointer to buffer holding message       */
int len;        /* length value in bytes being send        */
int node;       /* value of the destination node           */
int pid;        /* value of the destination process id     */
{
   int *malloc();
   void free();

   int  *send_buf_ptr;
```

```
int  *ovr_head_ptr;
char *temp1_ptr;
char *temp2_ptr;
int i, j;
int out_len;

static int initialized    = 0;
static int my_node, my_pid, cube_dim;
static int over_head_size;
static int nearest_nodes[MAX_CUBEDIM];
if (!initialized)
   {
   my_node  = mynode();
   my_pid   = mypid();
   cube_dim = cubedim();
   over_head_size = OVR_HEAD * sizeof(int);

   for (i=0; i < cube_dim; i++)
      {
      j = 1 << i;
      nearest_nodes[i] = my_node ^ j;
      }
   initialized = 1;
   }

/*********************************************************
*  Code to allow direct transmission of a message to  *
*  itself, a neighbor node, or make a global          *
*  transmission without using the routing routine.    *
*********************************************************/
for(i=0; node != nearest_nodes[i] && i < cube_dim; i++)
   ;

if(node == my_node || node < 0 || i < cube_dim)
   {
   send(ci, type, buf_ptr, len, node, pid);
   while(status(ci)) flick();
   }
else
   {
```

```
        /*****************************************************
        *  Code to save routing overhead in first six       *
        *  integers of message.                             *
        *****************************************************/
        out_len      = over_head_size + len;
        ovr_head_ptr = malloc(out_len);
        send_buf_ptr = ovr_head_ptr;

        *ovr_head_ptr++  = node;
        *ovr_head_ptr++  = pid;
        *ovr_head_ptr++  = type;
        *ovr_head_ptr++  = my_node;
        *ovr_head_ptr++  = my_pid;
        *ovr_head_ptr++  = len;

        /*************************************************
        *  Code to copy message into new data buffer.  *
        *************************************************/
        temp1_ptr = (char *)ovr_head_ptr;
        temp2_ptr = buf_ptr;

        for(i=0; i < len; i++)
           *temp1_ptr++ = *temp2_ptr++;

        send(ci, ROUTE_TYPE, send_buf_ptr,
           out_len, my_node, ROUTE_PID);

        while(status(ci)) flick();

        free(send_buf_ptr);

        }

    }  /* End of asend function   */
```

```
/*************************************************************
*                                                            *
*  DATE:  09/20/87                                           *
*  VERSION:  2.2                                             *
*                                                            *
*  TITLE: Non-Blocking Receive for Adaptive Routing Process  *
*  FILENAME:  arecv.c                                        *
*  PROJECT:  Thesis                                          *
*  OPERATING SYSTEM:                                         *
*  LANGUAGE: C                                               *
*  FILE PROCESSING: compiled with options -ALfu              *
*  FUNCTION:  This fuction can be linked to any program      *
*  using the follwing parameters:                            *
*     int ci;                                                *
*     int type;                                              *
*     char buf[n];                                           *
*     int len;                                               *
*     int cnt;                                               *
*     int node;                                              *
*     int pid;                                               *
*                                                            *
*************************************************************/

#define ROUTE_PID  32767
#define ROUTE_TYPE 32767

#define OVR_HEAD  6

arecv(ci, type, buf_ptr, len, pt_cnt, pt_node, pt_pid)

int ci;         /* channel value from calling programming   */
int type;       /* type value from calling programming      */
char *buf_ptr; /* pointer to calling program buffer space  */
int len;        /* length value of buffer from calling prg  */
int *pt_cnt;    /* pointer to count of rec'd message        */
int *pt_node;   /* pointer to source node of rec'd message  */
int *pt_pid;    /* pointer to source proc. id of rec'd mesg */

{
   int *malloc();
   void free();
```

```
        int i, buf_size;
        int *ovr_head_ptr;
        int *recv_buf_ptr;
        char *temp1_ptr;
        char *temp2_ptr;
        static int initialized = 0;
        static int over_head_size;

        if (!initialized)
           {
           over_head_size = OVR_HEAD * sizeof(int);
           initialized    = 1;
           }

        buf_size     = over_head_size + len;
        recv_buf_ptr = malloc(buf_size);
        recv(ci, type, recv_buf_ptr, buf_size,
           pt_cnt, pt_node, pt_pid);

        while(status(ci)) flick();

        if (*pt_pid == ROUTE_PID)
           {
           ovr_head_ptr = recv_buf_ptr + 3;
           *pt_node = *ovr_head_ptr++;
           *pt_pid  = *ovr_head_ptr++;
           *pt_cnt  = *ovr_head_ptr++;
           }

        else
           ovr_head_ptr = recv_buf_ptr;

        temp1_ptr = (char *)ovr_head_ptr;
        temp2_ptr = buf_ptr;

        for(i=0; i < *pt_cnt; i++)
           *temp2_ptr++ = *temp1_ptr++;

        free(recv_buf_ptr);
        return;
     }  /* End of arecv function  */
```

Appendix C. *Host Process for Adaptive Routing Testing*

```
/*************************************************************
*                                                            *
*  DATE:  09/20/87                                           *
*  VERSION:  2.2   Based on iPSCs Host Process for Ring Ex.  *
*                                                            *
*  TITLE:  Sequential Ring Host Process                      *
*  FILENAME:  host.c                                         *
*  PROJECT:  Thesis                                          *
*  OPERATING SYSTEM:  XENIX                                  *
*  LANGUAGE: C                                               *
*  FILE PROCESSING: compiled with options -ALfu              *
*  FUNCTION:  This is the Host code for the ring demo.       *
*  It loads 3 processes:                                     *
*     a) the routing process                                 *
*     b) the node process                                    *
*     c) the loading process                                 *
*  It reads 2 files for testing input:                       *
*     a) Information for the load process                    *
*        1) the number of links to be congested              *
*        2) the length of the message for the load process   *
*     b) Information for the node process                    *
*        1) the number of times to go around the RING.       *
*        2) the length of the message in bytes               *
*  It opens 1 file for output:                               *
*     a) the number of links congested                       *
*     b) the length of the message in the load process       *
*     c) a ring "count" each time the ring message           *
*        goes past node 0,                                   *
*     d) the time it took the message to go around           *
*        the ring the specified number of times.             *
*     e) the average time per pass through the network       *
*                                                            *
*************************************************************/
#include "/usr/ipsc/lib/chost.def"
#include <stdio.h>

/*************************************************************
*  Program definitions and varibles.                         *
*************************************************************/
```

```
#define HOST_PID   1
#define NODE_PID   2
#define ROUTE_PID  32767
#define LOAD_PID   3010

#define ALL_NODES -1
#define ALL_PIDS  -1

#define INIT_TYPE 10

#define INIT_MSG_SIZE 4
#define CNT_MSG_SIZE  2
#define TIME_MSG_SIZE 4
#define MAX_MSG_SIZE  16384

#define ROUTE_TEST      0

/************************
*  Program variables:  *
************************/
int ci, type;
int cnt, fr_node, fr_pid;
int i, j, ring_count;
int msg_buff[8192];
int msg_len;
int num_links, msg_load;

long   time_buff;
float  ring_time;

char CARRIAGE_RETURN = 13;

FILE *fp_in, *fp_out, *fp_load;

/*************************************************************/
main()
{

   printf("LOADING THE CUBE WITH ROUTE ... ");
   printf("ONE MOMENT PLEASE\n");
   load("route", ALL_NODES, ROUTE_PID);
   printf("LOADING THE CUBE WITH NODE  ... ");
```

```
printf("ONE MOMENT PLEASE\n");
load("node", ALL_NODES, NODE_PID);

/*********************************************************
*  Open a channel for the host-to-node communications.   *
*********************************************************/
ci = copen(HOST_PID);

/*********************************************************
*  Open node input and test data output files.           *
*********************************************************/
fp_load =
   fopen("/usr/eng/tfarinel/mydata/load_data","r");
fp_out =
   fopen("/usr/eng/tfarinel/a_routing/out_data","w");

/*********************************************************
*  BEGIN MAIN PROGRAM LOOP TO CONTROL LOAD PROCESS:      *
*********************************************************/
for(;;)
   {
   printf("*************** START LOAD *************\n");

   /*****************************************************
   *  get the number of links for the load process:     *
   *****************************************************/

   printf("Number links for load process ");
   printf("(neg. value quits): ");
   fscanf (fp_load,"%d", &num_links);
   printf("%d\n",num_links);

   /*****************************************************
   *  If num_links is < 0, break out & clean up:        *
   *****************************************************/
   if (num_links < 0) break;

   /*****************************************************
   *  get the number of bytes in the load message:      *
   *****************************************************/
   printf("Number of bytes in the message (0-16384): ");
   fscanf(fp_load,"%d", &msg_load);
```

```
printf("%d\n",msg_load);
/*******************************************************
*  Include num_links and message length in the         *
*  message to the load process:                        *
*******************************************************/
msg_buff[0] = num_links;
msg_buff[1] = msg_load;

if (num_links != 0)
   {
   printf("LOADING THE CUBE WITH LOAD  ...");
   printf(" ONE MOMENT PLEASE\n");
   load("myload", ALL_NODES, LOAD_PID);

   /*****************************************************
   *  Send the message buffer to all the nodes:         *
   *****************************************************/
   sendmsg(ci, INIT_TYPE, msg_buff,
      INIT_MSG_SIZE, ALL_NODES, LOAD_PID);
   }

fprintf(fp_out,"\n Number  Message  Ring");
fprintf(fp_out,"   Message  Total  Average\n");
fprintf(fp_out," Links   Length   Count  Length");
fprintf(fp_out,"   Time   Time\n\n");
/*******************************************************
*  Open node process input data files.                 *
*******************************************************/
fp_in  =
   fopen("/usr/eng/tfarinel/mydata/in_data","r");

/*******************************************************
*  BEGIN MAIN LOOP TO CONTROL NODE PROCESS:            *
*******************************************************/
for(;;)
   {
   printf("*************** START RING **********\n");

   /*****************************************************
   *  get the number of times to go around the ring:    *
   *****************************************************/
   printf("Number of times to go around the ring ");
```

```
printf("(neg. value quits): ");
fscanf (fp_in,"%d", &ring_count);
printf("%d\n",ring_count);

/*****************************************************
*  If ring_count is negative break out of loop.     *
*****************************************************/
if (ring_count < 0) break;

/*****************************************************
*  get the number of bytes in the message:          *
*****************************************************/
printf("Number of bytes in the message (0-16372):");
fscanf(fp_in," %d", &msg_len);
printf(" %d\n",msg_len);

/*****************************************************
*  Include ring_count and message length in the     *
*  message to the ring process:                     *
*****************************************************/
msg_buff[0] = ring_count;
msg_buff[1] = msg_len;

/*****************************************************
*  Send the message buffer to node 0:               *
*****************************************************/
sendmsg(ci, INIT_TYPE, msg_buff,
   INIT_MSG_SIZE, 0, NODE_PID);

/*****************************************************
*  Get the current ring count from node 0           *
*  and report to user:                              *
*****************************************************/
for (i=1;i<=ring_count;i++)
   {
   recvmsg(ci, &type, msg_buff,
      CNT_MSG_SIZE, &cnt, &fr_node, &fr_pid);
   printf("Ring count: %d %c", msg_buff[0],
      CARRIAGE_RETURN);
   }
```

```
/*****************************************************
*  Get the RING time from node 0 & report to user:  *
*****************************************************/
recvmsg(ci, &type, &time_buff,
   TIME_MSG_SIZE, &cnt, &fr_node, &fr_pid);
ring_time = (float)time_buff/1000.00;
printf("\nRing time : %0.2f secs.\n", ring_time);
fprintf(fp_out,"%4d %9d %6d %8d  %7.2f  %7.2f\n",
   num_links, msg_load, ring_count, msg_len,
   ring_time, ring_time/ring_count);

/*****************************************************
*  Used to get path data.                           *
*  Not used in time testings.                       *
*****************************************************/
#if (ROUTE_TEST)
   {
   recvmsg(ci, &type, msg_buff, MAX_MSG_SIZE,
      &cnt, &fr_node, &fr_pid);
   for(i=0; msg_buff[i] >= 0; i++)
      {
      if(msg_buff[i] == 100)
         {
         fprintf(fp_out,"\n\n");
         j=0;
         }
      fprintf(fp_out,"%5d%c",msg_buff[i],
         (j%10 == 9) ? '\n' : ' ');
      j += 1;
      }
   fprintf(fp_out,"\n\n");
   }
#endif

   }  /* END OF MAIN PROGRAM LOOP FOR NODE PROCESS.  */

fclose("/usr/eng/tfarinel/mydata/in_data");
lkill(ALL_NODES,LOAD_PID);
lwaitall(ALL_NODES,LOAD_PID);

}  /*  END OF MAIN PROGRAM LOOP FOR LOAD PROCESS.    */
```

```
    /*************************************************************
    *  CLEAN UP TIME!                                            *
    *  Close data files.                                         *
    *************************************************************/
    fclose("/usr/eng/tfarinel/mydata/load_data");
    fclose("/usr/eng/tfarinel/a_routing/out_data");

    /*************************************************************
    *  Kill RING processes in cube:                              *
    *************************************************************/
    printf("CLEARING THE CUBE ...\n");

    lkill(ALL_NODES,ALL_PIDS);
    lwaitall(ALL_NODES,ALL_PIDS);

    printf("******************** DONE ******************\n");

}   /* END OF HOST PROGRAM                                       */
```

Appendix D. *Ring Control Process for Adaptive Routing Testing*

```
/*************************************************************
*                                                            *
*  DATE:  09/21/87                                           *
*  VERSION:  2.2   Based on Intel's example RING program.    *
*                                                            *
*  TITLE:  Sequential Node Ring Process                      *
*  FILENAME:  node.c                                         *
*  PROJECT:  Thesis                                          *
*  OPERATING SYSTEM:  XENIX                                  *
*  LANGUAGE: C                                               *
*  FILE PROCESSING: compiled with options -ALfu              *
*  FUNCTION:  This process passes variable length messages   *
*  from node 0 to all other nodes in the iPSC before         *
*  returning to node 0.  Each node increments the node       *
*  address by modulo(cubedim()).  For cubedim of 3, the      *
*  path taken is 0-1-2-3-4-5-6-7-0.                          *
*                                                            *
*  Node 0 will play the role of "controller" node.           *
*                                                            *
*  It waits for a message from the host telling it:          *
*     a) the number of times to go around the RING.          *
*                                                            *
*  It then sends a message to node 1 and counts the times    *
*  the message goes around the RING.                         *
*                                                            *
*  At the end, Node 0 reports back to the Host the time it   *
*  took the message to go around the RING.                   *
*                                                            *
*************************************************************/
#include "/usr/ipsc/lib/cnode.def"

#define HOST_NID  0x8000
#define HOST_PID  1
#define INIT_TYPE  10
#define NODE_TYPE  20
#define TIME_TYPE  30
#define COUNT_TYPE 40

#define INIT_SIZE       4
```

```
#define TIME_SIZE        4
#define COUNT_SIZE       2
#define MAX_MSG_SIZE 16384
#define ROUTE_TEST  0

/***********************
*  Program variables:  *
***********************/
int host_chan, node_chan;
int i, count, ring_count;
int msg_len;
int init_buff[2];
int msg_buff[8192];
int my_node, my_pid;
int next_node, next_pid;
int num_nodes;
int rcnt, rnode, rpid;
int limit, j;                /* Not used in time testing  */

long start_time, ring_time;

/*************************************************************/

main()
{

   /*****************************************
   *  Each process identifies the node its  *
   *  running on and its pid:               *
   *****************************************/
   my_node = mynode();
   my_pid  = mypid();

   /*****************************************
   *  Each process determines the node id   *
   *  & and the pid of the node following   *
   *  itself in the RING:                   *
   *****************************************/
   num_nodes = 1<<cubedim();
   next_node = (my_node + 1)% num_nodes;
   next_pid  = my_pid;
```

```
/*****************************************
*  Open channel for communicating with   *
*  the next node in the RING.            *
*****************************************/
node_chan = copen(my_pid);

/*****************************************
*  BEGIN NODE 0 CODE:                    *
*****************************************/
if (my_node == 0)
   {

   /*************************************
   *  Open channel for communicating    *
   *  with the host.                    *
   *************************************/
   host_chan = copen(my_pid);

   /*************************************
   *  NODE 0 MAIN LOOP:                 *
   *************************************/
   for (;;)
      {
      recvw(host_chan, INIT_TYPE, init_buff,
         INIT_SIZE, &rcnt, &rnode, &rpid);

      ring_count = init_buff[0];
      msg_len    = init_buff[1];

      /*********************************
      *  Used during path testing.     *
      *  Not used during time testing. *
      *  Initialize message to all -1's.*
      *********************************/
      #if (ROUTE_TEST)
         {
         limit = (msg_len / 2) + 1;

         for(i=0; i < limit; i++)
            msg_buff[i] = -1;
         }
      #endif
```

```
        ring_time = 0;

        for(i=1;i<=ring_count;i++)
           {
           #if (ROUTE_TEST)
              {
              for(j=0; msg_buff[j] >= 0; j++)
                 ;
              msg_buff[j] = my_node + 100;
              }
           #endif

           start_time = clock();

           /*****************************************
           *  Use routing interface calls asendw()  *
           *  and arecvw() instead of sendw() and   *
           *  recvw() as defined in iPSC manual.    *
           *****************************************/
           asendw(node_chan, NODE_TYPE, msg_buff,
              msg_len, next_node, next_pid);

           arecvw(node_chan, NODE_TYPE, msg_buff,
              MAX_MSG_SIZE, &rcnt, &rnode, &rpid);

           ring_time += (clock() - start_time);
           count = i;

           sendw(host_chan, COUNT_TYPE, &count,
              COUNT_SIZE, HOST_NID, HOST_PID);
           }

        sendw(host_chan, TIME_TYPE, &ring_time,
           TIME_SIZE, HOST_NID, HOST_PID);

        #if (ROUTE_TEST)
           sendw(host_chan, NODE_TYPE, msg_buff,
              msg_len, HOST_NID, HOST_PID);
        #endif
        }
     }
```

```
        else
           {
           /**********************************
           *  BEGIN OTHER NODES' MAIN LOOP:  *
           **********************************/
           for (;;)
              {
              arecvw(node_chan, NODE_TYPE, msg_buff,
                 MAX_MSG_SIZE, &rcnt, &rnode, &rpid);

              #if (ROUTE_TEST)
                 {
                 for(j=0; msg_buff[j] >= 0; j++)
                    ;
                 msg_buff[j] = my_node + 100;
                 }
              #endif

              asendw(node_chan, NODE_TYPE, msg_buff,
                 rcnt, next_node, next_pid);
              }
           }
     } /*  End of Sequential Node Ring Process    */
```

Appendix E. *Network Loading Process for Adaptive Routing Testing*

```
/*************************************************************
*                                                            *
*  DATE:  09/20/87                                           *
*  VERSION:  2.2                                             *
*                                                            *
*  TITLE:  Load Process for Adaptive Routing Process         *
*  FILENAME:  myload.c                                       *
*  PROJECT:  Thesis                                          *
*  OPERATING SYSTEM:  XENIX                                  *
*  LANGUAGE: C                                               *
*  FILE PROCESSING: compiled with options -ALfu              *
*  FUNCTION:  This process provides a communications load    *
*  to compare current iPSC routing and the distributed       *
*  routing process.  This loading process is loaded on all   *
*  nodes of the iPSC.  It causes congestion by forcing       *
*  additional messages to be passed between neighboring      *
*  nodes on the cube.                                        *
*                                                            *
*  The number of links and the message size to use is sent   *
*  from the "host" process.  This process determines         *
*  which nodes form the communication's bound links, based   *
*  on the value of the number of links parameter.  The       *
*  number of linnks must be an even number between           *
*  2 and 16, inclusive.                                      *
*                                                            *
*************************************************************/

#include "/usr/ipsc/lib/cnode.def"

#define INIT_TYPE    10
#define NODE_TYPE    30

#define MAX_SIZE  16384

/************************
*  Program variables:  *
************************/
int i, temp;
int node_chan, host_chan;
```

```
int msg_buff[MAX_SIZE/2];
int num_links, msg_load;
int my_node, my_pid;
int next_node;
int send_node = 0;
int recv_node = 0;
int rcnt, rnode, rpid;

/*************************************************************
*  Beginning of main routing, that waits for a message from *
*  the host specifing the number of links to be congested   *
*  and the message length to be used.  This also allows the *
*  host program to determine when the loading should begin. *
*************************************************************/

main()
{

   /*************************************
   *  Each process identifies its node  *
   *  & its pid:                        *
   *************************************/
   my_node = mynode();
   my_pid  = mypid();

   host_chan = copen(my_pid);

   recvw(host_chan, INIT_TYPE, msg_buff,
      MAX_SIZE, &rcnt, &rnode, &rpid);
   num_links = msg_buff[0];
   msg_load  = msg_buff[1];

   /*****************************************
   *  Determine sending nodes and receiving  *
   *  nodes for the loading process.         *
   *****************************************/
   for (i=0; i < num_links/2; i++)
      {
      temp = i * 4;
      if (my_node == temp || my_node == temp + 1)
         {
         send_node = 1;
```

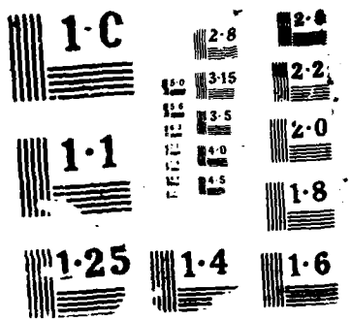
```
            next_node = my_node + 2;
            break;
            }
         else if (my_node == temp + 2 || my_node == temp + 3)
            {
            recv_node = 1;
            break;
            }
         else
            ;
         }

      /*****************************************
      *  Open channel for communicating with the  *
      *  next node in the RING.                   *
      *****************************************/
      node_chan = copen(my_pid);

      /*****************************************
      *  Each node determines to send, receive,  *
      *  or allow other processes to continue.   *
      *****************************************/
      for (;;)
         {
         if (send_node)
            {
            sendw(node_chan, NODE_TYPE, msg_buff,
               MAX_SIZE, next_node, my_pid);
            recvw(node_chan, NODE_TYPE, msg_buff,
               MAX_SIZE, &rcnt, &rnode,  &rpid);
            }
         else if (recv_node)
            {
            recvw(node_chan, NODE_TYPE, msg_buff,
               MAX_SIZE, &rcnt, &rnode, &rpid);
            sendw(node_chan, NODE_TYPE, msg_buff,
               rcnt, rnode, my_pid);
            }
         else
            flick();
   }  /* End of myload function */
```

AD-A189 849 IMPLEMENTATION OF A DISTRIBUTED ADAPTIVE ROUTING ALGORITHM ON THE INTEL I.. (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T C FARINELLI 2/2

UNCLASSIFIED DEC 87 AFIT/GCS/ENG/87D-11 F/G 12/5 NL


1·C
2·8
3·15
3·5
4·0
4·5
2·2
2·0
1·8
1·1
1·25
1·4
1·6

Appendix F. *Makefile for Adaptive Routing Processes*

```
CFLAGS = -Alfu -K

#
#  NOTE:  This makefile uses the default rule for the C
#         compiler for node.c
#

all: host node route myload
rtest:    host node route

help:
     @echo "make all    - makes all processes"
     @echo "make host   - makes the host process"
     @echo "make node   - makes the node process"
     @echo "make clean  - cleans up"


host:     host.c
     cc -Alfu -o host host.c /usr/ipsc/lib/chost.a


node:     node.o
     ld -Ml -o node /lib/Lseg.o /usr/ipsc/lib/Lcrtn0.o  \
     node.o   \
     /usr/eng/tfarinel/lib/obj/asendw.o                 \
     /usr/eng/tfarinel/lib/obj/arecvw.o                 \
     /usr/ipsc/lib/Llibcnode.a                          \
     /usr/ipsc/lib/Llibcel.a                            \
     /usr/intel/lib/cel287.a


route:  route.o
     ld -Ml -o route /lib/Lseg.o /usr/ipsc/lib/Lcrtn0.o \
     route.o                                            \
     /usr/ipsc/lib/Llibcnode.a                          \
     /usr/ipsc/lib/Llibcel.a                            \
     /usr/intel/lib/cel287.a
```

```
myload: myload.o
     ld -Ml -o myload /lib/Lseg.o /usr/ipsc/lib/Lcrtn0.o \
     myload.o                                            \
     /usr/ipsc/lib/Llibcnode.a                           \
     /usr/ipsc/lib/Llibcel.a                             \
     /usr/intel/lib/cel287.a

clean:
     -rm *.o *log
```

Bibliography

1. *iPSC System Overview (Vol 1).* Intel Scientific Computers., Beaverton, Oregon. November 1986.
2. *iPSC System Product Summary.* Intel Scientific Computers., Beaverton, Oregon.
3. *Microcommunications Handbook.* Intel Corporation., Santa Clara, California. 1986.
4. Kenneth Brayer. Survivable Computer Communication Routing Using Decentralized Control. *IEEE International Conference on Circuits and Computers.* ICCC 82:93–97, September 1982.
5. Joe Edmond Defenderfer. *Comparative Analysis of Routing Algorithms for Computer Networks.* Master's thesis, ESL-R-756. Cambridge Electronic Systems Lab, Massachusetts Institute of Technology, Cambridge MA, June 1977. (AD-A041 293).
6. Tse-yun Feng. A survey of interconnection networks. *IEEE Transaction on Computers*, 109–110, December 1981.
7. Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE.* Vol. 54, 12:1901–1909, December 1966.
8. Geoffrey C. Fox. The caltech concurrent computation program. In *Hypercube Multiprocessors 1987*, pages 353–379, SIAM, Philadelphia, 1987.
9. Albert B. Garcia. *Introduction to Computer Networks.* Technical Report. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, October 1986. EENG 554, Class Handout.
10. James M. Kasson and Richard S. Kagan. *Data Communications, Networks, and Systems*, chapter PBX Local Area Networks, pages 144–145. Howard W. Sams & Co., Indianapolis, 1985.
11. John M. McQuillan, Ira Richer, and Eric C. Rosen. The new routing algorithm for the ARPANET. *IEEE Transactions on Communications*, COM-28:711–719, May 1980.
12. Justin Rattner. Concurrent processing: a new direction in scientific computing. In *AFIPS 1985 National Computer Conference*, pages 157–166, July 1985.
13. Harry Rudin. On routing and "delta routing": a taxonomy and performance comparison of techniques for packet-switched networks. *IEEE Transactions on Communications*, COM-24:43–59, January 1976.
14. Howard J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing.* Lexington Books, Lexington, 1985.

15. J.M. Spinelli. *Broadcasting Topology and Routing Information in Computer Networks.* Master's thesis, LIDS-TH-1470. Cambridge Lab for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge MA, May 1985. (AD-A155 668).

16. John A. Stankovic. A perspective on distributed computer systems. *IEEE Transaction on Computers*, C-33:1102–1115, December 1984.

17. Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs. 1981.

18. Stuart Wecker. DNA: the digital network architecture. *IEEE Transactions on Communications*, COM-28:510–526, April 1980.

Vita

Captain Tommy C. Farinelli was born on 20 May 1952 in Butler, Pennsylvania. He graduated from high school in Slippery Rock, Pennsylvania, in 1970 and enlisted in the USAF in 1971. In 1980, he was selected for the Airmen's Education and Commissioning Program and attended Auburn University, Alabama, from which he received the degree of Bachelor of Computer Engineering in 1983. Upon graduation, he attended the Officer's Training School where he received his commission in the USAF. He then served as a Test Manger for Joint Tactical Systems in the 4501st Computer Services Squadron and the 1912th Information System Services Group, Langley AFB, Virginia, until entering the School of Engineering, Air Force Institute of Technology, in June 1986.

Permanent address: Box 204
Prospect, *Pennsylvania*
16052

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/87D-11	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: CSD/SDIO; 8b. OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): ATTN: S/BM LtCol Sowa, Pentagon, Washington DC 20301-7100	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
IMPLEMENTATION OF A DISTRIBUTED ADAPTIVE ROUTING ALGORITHM ON THE INTEL iPSC (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Tommy C. Farinelli, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1987 December	102

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Routing, Communications Networks, Distributed Data Processing
12	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: Charles R. Bisbee, ItCol, USAF

Lynn Wolaver 31 Dec 87
Development
(AFIT)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Charles R. Bisbee, ItCol, USAF	513-255-3576	AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

19. Abstract

The purpose of this study was to examine the use of distributed adaptive routing algorithms on concurrent class computers. The Intel Personal SuperComputer (iPSC) was used as the test computer system. The implemented routing algorithm allowed each node to select the next node based on two criteria. The first criteria was the fewest number of hops, the second was the smallest delay time.

This study was limited to the comparison of a distributed adaptive routing algorithm, implemented at the applications layer, with the current static routing and with a simulation of the current routing implemented at the applications layer. The comparsion with the current routing algorithm provides a measure of the penalty for the implementation at the applications layer. The comparsion with the simulated current static routing provides a measure of the possible performance gain had the adaptive routing algorithm been implemented at the network layer.

In all three configurations were tested to formulate the comparisons. Each configuration was comprised of four processes: a Host Process, a Routing Process, a Ring Control Process, and a Network Loading Process. The Host Process controlled the loading of the processes onto the iPSC, the Routing Process controlled the message routing, the Ring Control Process provided the baseline message passing, while the Network Loading Process provided communications congestion on selected links. The metric used to compare the Routing Process performance was the average delay time for passing a message around the ring.

END

DATE

FILMED

APRIL

1988

DTIC